THE EXPERT'S VOICE® IN WEB DEVELOPMENT

Beginning JavaScript Charts

With jqPlot, D3, and Highcharts

DISPLAY YOUR DATA USING INNOVATIVE BROWSER-BASED CHARTS

Fabio Nelli

Apress®

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.

Apress®

Contents at a Glance

About the Author xix

About the Technical Reviewer xxi

Acknowledgments xxiii

Introduction xxv

■ Chapter 1: Charting Technology Overview 1

■ Chapter 2: jQuery Basics 19

■ Chapter 3: Simple HTML Tables 43

■ Chapter 4: Drawing a Line Chart 61

■ Chapter 5: Drawing a Bar Chart 81

■ Chapter 6: Drawing a Pie Chart 85

■ Chapter 7: Creating a Library for Simple Charts 113

■ Chapter 8: Introducing jqPlot 131

■ Chapter 9: Line Charts with jqPlot 151

■ Chapter 10: Bar Charts with jqPlot 221

■ Chapter 11: Pie Charts and Donut Charts with jqPlot 257

■ Chapter 12: Candlestick Charts with jqPlot 267

■ Chapter 13: Scatter Charts and Bubble Charts with jqPlot 273

■ Chapter 14: Funnel Charts with jqPlot 283

■ Chapter 15: Adding Controls to Charts 287

■ Chapter 16: Embedding jqPlot Charts in jQuery Widgets 303

Chapter 17: Handling Input Data 319
Chapter 18: Moving from jqPlot to Highcharts 329
Chapter 19: Working with D3 373
Chapter 20: Line Charts with D3 401
Chapter 21: Bar Charts with D3 449
Chapter 22: Pie Charts with D3 481
Chapter 23: Candlestick Charts with D3 503
Chapter 24: Scatterplot and Bubble Charts with D3 513
Chapter 25: Radar Charts with D3 545
Chapter 26: Handling Live Data with D3 557
Appendix A: Guidelines for the Examples in the Book 573
Appendix B: jqPlot Plug-ins 581

Index 583

Introduction

Welcome to the world of charts. If you are holding this book in your hands, you are undoubtedly interested in data visualization, perhaps with the hope of developing web pages filled with interactive charts. Or, maybe your purpose is to improve your knowledge of the jqPlot, D3, or Highcharts library. Whatever your objective, I hope this book enables you to achieve it.

In addition to the various types of charts and JavaScript libraries, this book covers a range of topics: the jQuery library and selections, HTML5 and the canvas, widgets and controls, graphic manipulation with scalable vector graphics (SVG) technology, and mathematical concepts (scales and domains, curve fitting and trend lines, clustering analysis, and much more).

I have enriched this wide range of topics with many examples, each tightly focused on a particular one and presented to you in an ordered sequence, with step-by-step instructions.

Chart development can be easy once you know the process and have the right tools at the ready. Therefore, in presenting this material, I have included helpful, reusable code snippets as well as explanations of underlying concepts. After reading this book, you will be equipped to create any type of data visualization, either traditional or newer, with confidence.

CHAPTER 1

Charting Technology Overview

When we need to represent data or qualitative structures graphically in order to show a relationship—to make a comparison or highlight a trend—we make use of charts. A chart is a graphic structure consisting of symbols, such as lines, in a line chart; bars, in a bar chart; or slices, in a pie chart. Charts serve as valid tools that can help us discern and understand the relationships underlying large quantities of data. It is easier for humans to read graphic representations, such as a chart, than raw numeric data. Nowadays, use of charts has become common practice in a wide variety of professional fields as well as in many other aspects of daily life. For this reason, charts have come to take on many forms, depending on the stucture of the data and the phenomenon that is being highlighted. For example, if you have data separated into different groups and want to represent the percentage of each, with respect to the total, you usually display these groups of data in a pie chart or a bar chart. In contrast, if you want to show the trend of a variable over time, a line chart is typically the best choice.

In this book, you will learn how to create, draw, and adapt charts to your needs, using various technologies based on JavaScript. Before you start using JavaScript to develop charts, however, it is important that you understand the basic concepts that will be covered in the chapters of this book. In this chapter, I will provide a brief overview of these concepts.

First, I will show you how to recognize the most common elements that make up a chart. Knowledge of these elements will prove helpful, because you will find them in the form of components, variables, and objects defined within the specialized JavaScript libraries created for the realization of charts.

Next, I will present a list of the most common types of charts. The greater your knowledge of charts and their features, the easier it will be to choose the right representation for your data. Making the right choice is essential if you are to underline the relationships you want to represent, and just reading the data will not be sufficent. Only when you have become familiar with the most common types of charts will you be able to choose which is the most suitable for your purposes.

Once you have become familiar with these concepts, you will need to learn how it is possible to realize them via the Web and what the current technologies are that can help you achieve this aim. Thus, in the second part of the chapter, I will discuss these technical aspects, presenting one by one the technologies involved in the development of the examples provided in this book.

Finally, given that all our work will focus on the development of code in JavaScript, I thought it would be helpful to provide a brief description of certain types of data. Those who are not familiar with JavaScript can benefit from this quick reference source on the forms that the data will take within the code. However, I strongly recommend that the reader research in greater depth the concepts and technologies discussed in this chapter.

Elements in a Chart

As you will soon see, charts can assume a variety of forms. In a chart the data take on graphic structure through the use of symbols specific to the type of chart; there are, however, some features that are common to all charts.

Generally, every chart has a title, appearing at the top, that provides a short description of the data. Less frequently, subtitles or footnotes are used to supply additional descriptions (mostly data-related information, such as references, places, dates, and notes).

Charts often have axes—two perpendicular lines that allow the user to refer to the values of the coordinates (x, y) for each data point P(x, y), as shown in Figure 1-1. The horizontal line usually represents the x axis, and the vertical line, the y axis.

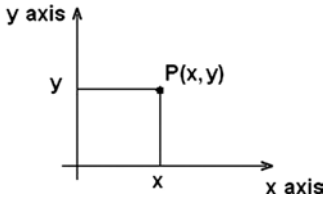


Figure 1-1. *A two-dimensional chart*

A scale is defined on each axis. The scale can be either numerical or categorical. Each axis is divided into segments corresponding to the particular range of values represented by the scale. The boundaries between one segment and the next are called ticks. Each tick reports the value of the scale associated with that axis. Generally, call these tick labels.

Figure 1-2 shows four axes with different scales. Axes a and b have numerical scales, with a being a linear scale, and b, a logarithmic scale. Axes c and d have categorical scales, with c being ordinal and therefore following an ascending order, whereas d is only a sequence of categories without any particular order.

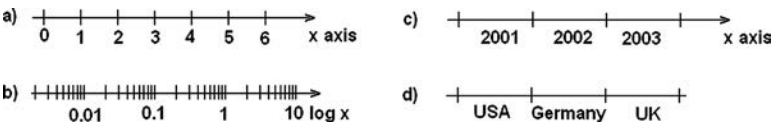


Figure 1-2. *Four axes with different scales*

Along with each axis, it is good practice to display a label briefly describing the dimension represented; these are called axis labels. If the scale is numerical, the label should show the units of measure in brackets. For instance, if you had an x axis reporting the timing for a set of data, you might write “time” as an axis label, with the second unit (in this case, seconds) in square brackets as [s] (see Figure 1-3).

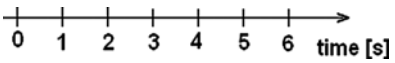


Figure 1-3. *An axis label*

In the drawing area displaying the chart, a line grid may be included to aid in the visual alignment of data. Figure 1-4 shows a grid for a chart with a linear time scale on the x axis and a logarithmic scale on the y axis.

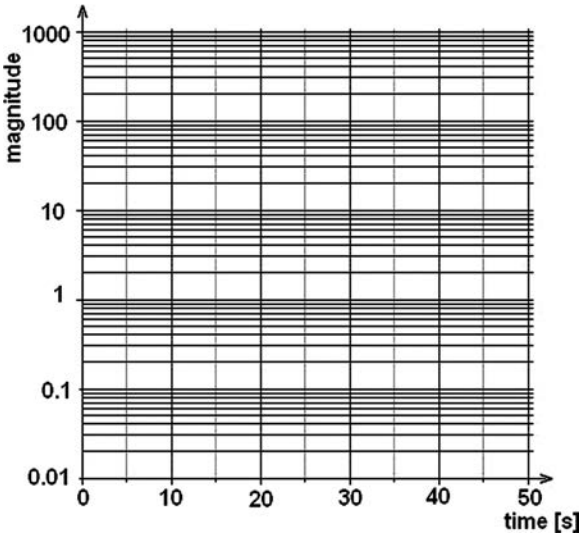


Figure 1-4. *A chart with two different scales*

You have seen how data can be represented symbolically. However, text labels can also be used to highlight specific data points. Point labels provide values in a chart right at the corresponding points in a chart, whereas tool tips are small frames that appear dynamically, when you pass the mouse over a given point. These two types of labels are shown in Figure 1-5.

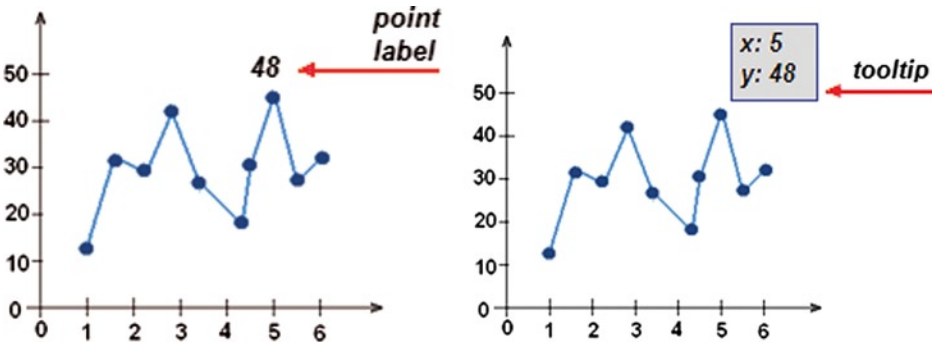


Figure 1-5. *The point label and the tooltip of a data point*

Data are often grouped in several series, and in order to represent these in the same chart, they must be distinguishable. The most common approach is to assign a different color to each series. In other cases, for example, with line charts, the line stroke (dashed, dotted, and so on) can also be used to distinguish different series. Once you have established a sequence of colors (or strokes), it is necessary to add a table demonstrating the correspondence between colors and groups. This table is called the legend and is shown in Figure 1-6.

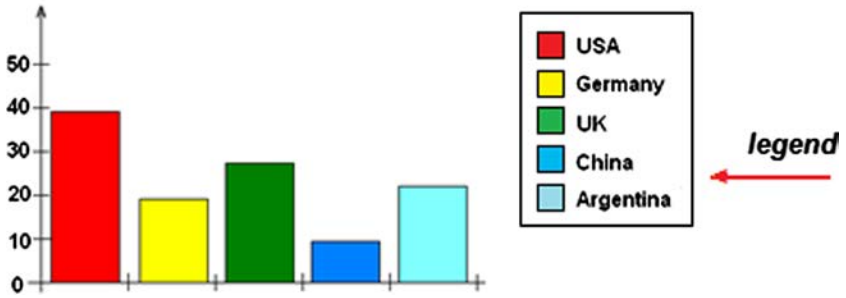


Figure 1-6. *A legend*

Although it may seem trivial to discuss the concepts covered in this section, it is important to define the terminology of the elements that I will be referring to throughout the book. They form the building blocks with which you will be building your charts. You will also see how JavaScript libraries specializing in the representation of charts use these terms, associating them with editing and setting components (see the section "Inserting Options" in Chapter 8).

Most Common Charts

This section contains a brief overview of the most common types of charts. These charts will each be described more thoroughly in the following chapters of the book.

Histogram: Adjacent rectangles erected on the x axis, split into discrete intervals (bins) and with an area proportional to the frequency of the observation for that bin (see Figure 1-7).

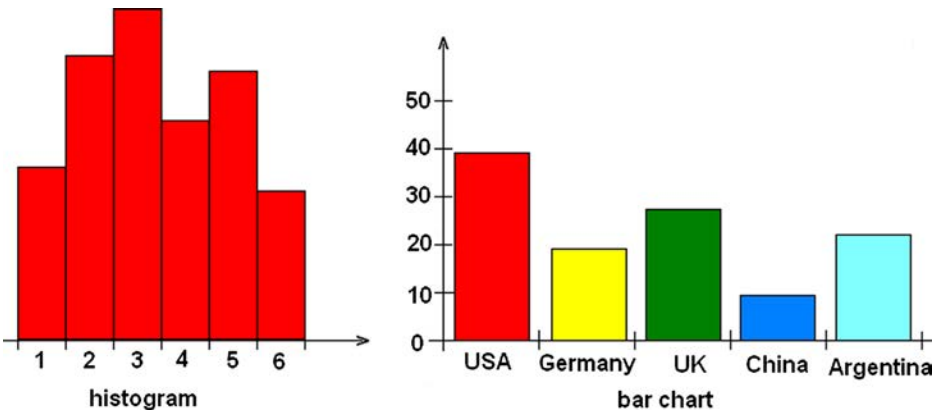


Figure 1-7. *A histogram and a bar chart*

Bar chart: Similar in shape to a histogram, but different in essence, this is a chart with rectangular bars of a length proportional to the values they represent. Each bar identifies a group of data (see Figure 1-7).

Line chart: A sequence of ordered data points connected by a line. Data points P(x, y) are reported in the chart, representing the scales of two axes, x and y (see Figure 1-8).

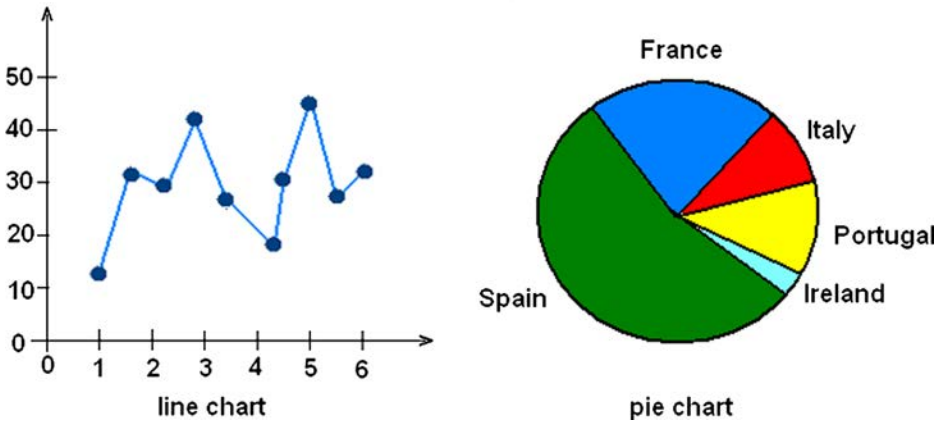


Figure 1-8. *A line chart and a pie chart*

Pie chart: A circle (pie) divided into segments (slices). Each slice represents a group of data, and its size is proportional to the percentage value (see Figure 1-8).

Bubble chart: A two-dimensional scatterplot in which a third variable is represented by the size of the data points (see Figure 1-9).

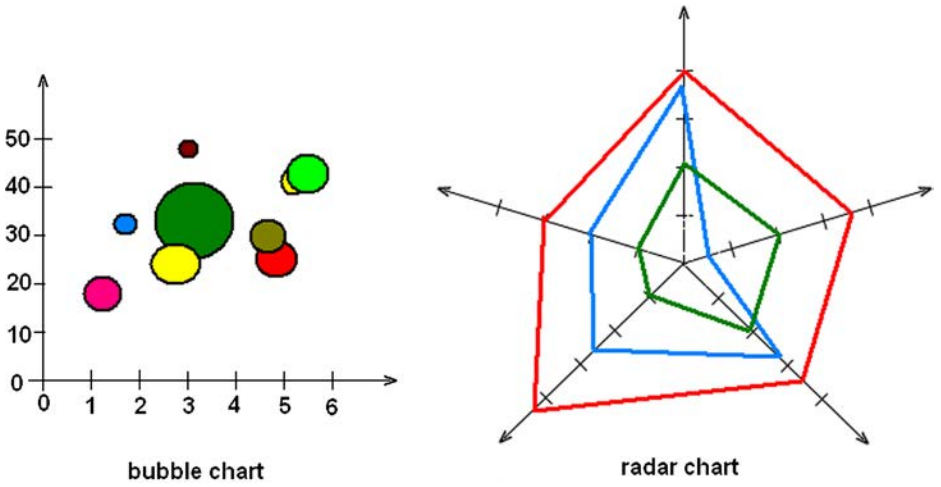


Figure 1-9. *A bubble chart and a radar chart*

Radar chart: A chart in which a series of data is represented on many axes, starting radially from a point of origin at the center of the chart. This chart often takes on the appearance of a spiderweb (see Figure 1-9).

Candlestick chart: A type of chart specifically used to describe price trends over time. Each data point consists of four values, generally known as open-high-low-close (OHLC) values, and assumes a shape resembling a candlestick (see Figure 1-10).

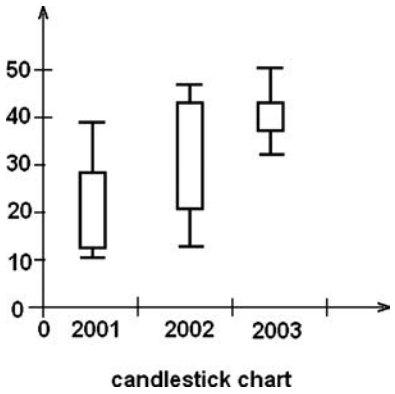


Figure 1-10. *A candlestick chart*

■ **Note** Open-high-low-close (OHLC) are four numeric values typically used to illustrate movement in the price of a financial instrument over time.

How to Realize Charts on the Web

Now that I have described the most common types of charts and the elements that compose them, the next step is to take a quick look at the technologies available today that will allow you to realize your chart.

Nowadays, web technologies are in constant flux: each day, new solutions are proposed, solving problems that only a short time ago would have proven to be quite complex. These innovations will afford you the possibility to realize highly interactive charts, with eye-catching graphics, all by writing only a few lines of JavaScript code. The whole thing can be done fast and easily, as most of the work is done for you, by the JavaScript libraries, which are highly specialized in chart representation. These libraries are now to be found all over the network.

In this book, you will work with jqPlot, Highcharts, and D3, which are currently the most widely used libraries and which can provide general solutions to practically any problem that may arise in the process of chart realization.

But, before stepping through these libraries one by one (which you will do in later chapters), you must first survey all the technologies that constitute the basis for chart development in JavaScript, as these will accompany you throughout the rest of the book.

HTML5

Recently, there has been a lot of talk about HTML5, which is practically revolutionizing the way in which web applications are developed. Before its advent, if you wanted to introduce interactive graphical content, the use of applications such as Adobe Flash was pretty much the obligatory path. But, dealing with Flash or similar applications for developing charts or other graphic representations on the Web involves an obvious limitation: dependency on a plug-in, to be installed on the end user's machine. In addition, these kinds of applications are not supported on smartphones. Thanks to HTML5, developers can now create advanced graphics and animation without relying on Flash.

As you read through this book, you will see how HTML5 has also led to the birth of many other technologies, some new, others old but renewed, such as JavaScript. In fact, as a language, JavaScript is experiencing a rebirth, as a result of the new libraries developed precicely to take advantage of the innovations introduced by HTML5. HTML5 has many new syntactical features, including the <canvas> elements and the integration of scalar vector graphics (SVG) content. Owing to these elements, it will be very easy to integrate multimedia and graphical content on the Web without using Flash.

In Flash's place, you will be using JavaScript libraries, such as jQuery, jqPlot, Highcharts, and D3. Currently, these are the most widespread and complete libraries available for the realization tasks such as the graphic visualization of data. The world of web technologies is constantly evolving, however; on the Internet, you can always find new libraries, with characteristics similar to those contained in this book.

Charting with SVG and CANVAS

Among all the possible graphic applications that can be implemented with the new technologies introduced by HTML5, I will focus on the representation and visualization of data through charts. Using JavaScript as a programming language, we can now take advantage of the powerful rendering engines embedded in new browsers. As the basis of the new capabilities of this language, I will refer to the HTML5 canvas and SVG. Instead of drawing static images on the server and then downloading them into the browser, SVG and canvas allow you to develop fully interactive charts and thus to enrich your graphic representations with built-in animation, transitions, and tool tips. This is because SVG and canvas content is drawn in the browser, and so the graphic elements that make up the chart can be transformed without refreshing the page. This feature is essential for visualizing real-time data, which require that the chart be continually updated, as the data change. Operating in this way will ensure a true client-side charting. In fact, by making use of these technologies, charts are actually drawn on the client and only require that the data be passed from the server. This aspect affords a considerable number of advantages, the foremost being elimination of the need for large graphics files to be downloaded from the server.

Canvas vs SVG

Both HTML5 canvas and SVG are web technologies that allow you to create rich graphics in the browser, but they are fundamentally different. Throughout this text, you will see mainly two JavaScript frameworks: **jqPlot** and **D3**. jqPlot, based on the jQuery framework, makes use of the HTML5 <canvas> element to draw its charts. In contrast, D3 does not make use of canvas; it relies on SVG technology for graphic representations.

SVG is an XML-based vector graphic format. SVG content can be static, dynamic, interactive, or animated, which makes it very flexible. You can also style the SVG elements with Cascading Style Sheets (CSS) and add dynamic behavior to them, using the application programming interface (API) methods provided by the SVG document object model (DOM). In choosing this format, you can, therefore, obtain much more than simple vector graphics and animation: you can develop highly interactive web applications, with scripting, advanced animation, events, filters, and almost anything else your imagination might suggest.

The HTML5 canvas specification is a versatile JavaScript API, allowing you to code programmatic drawing operations. Canvas, by itself, lets you define a canvas context object, shown as a <canvas> element on your HTML page. This element can then be drawn inside, using either a two-dimensional or three-dimensional drawing context, with Web Graphics Library (WebGL). I will cover only the first option; jqPlot uses a two-dimensional drawing context. The two-dimensional context provides you with a powerful API that performs quick drawing operations on a bitmap surface—the canvas. Unlike SVG, there are no DOM nodes for the shapes, only pixels.

The advantages of canvas, compared with SVG, are high drawing performance and faster graphics and image editing. Whenever it is necessary to work at the pixel level, canvas is preferable. However, with canvas, not having DOM nodes on which to work can be a disadvantage, especially if you do not use a JavaScript framework, such as jqPlot. Another disadvantage is poor text-rendering capabilities.

The advantages of SVG, compared with canvas, are resolution independence, good support for animation, and the ability to animate elements, using a declarative syntax.Most important, though, is having full control over each element, using the SVG DOM API in JavaScript. Yet, when complexity increases, slow rendering can be a problem, but browser providers are working hard to make browsers faster (see Tables 1-1 and 1-2).

Table 1-1. *Web Browsers and Engines*

Browser	Current	Engine	Developer	License
Google Chrome	29	Blink	Google, Opera, Samsung, Intel, others	GNU Lesser General Public License (LGPL), Berkeley Software Distribution (BSD) style
Mozilla Firefox	23	Gecko	Netscape/Mozilla Foundation	Mozilla Public License (MPL)
Internet Explorer	10	Trident	Microsoft	Proprietary
Apple Safari	6	WebKit	Apple, KDE, Nokia, Blackberry, Palm, others	GNU LGPL, BSD style

Table 1-2. *Web Technology Support: Comparison of Web Browsers*

	Browser			
Technology	Internet Explorer 10	Chrome 29	Firefox 23	Safari 6
SVG (v.1.1)				
Filters	Yes (from 10)	Yes	Yes	Yes (from 6)
Synchronized Multimedia Integration Language (SMIL) animation	No	Yes	Yes	Partial
Fonts	No	Yes	No	Yes
Fragment identifiers	Yes	No	Yes	No
HTML effects	Partial	Partial	Yes	Partial
CSS backgrounds	Yes	Yes	Partial	Yes
CSS	Yes	Yes	Yes	Yes
HTML5				
Canvas	Yes(from 9)	Yes	Yes	Yes
New elements	Yes	Yes	Yes	Yes
Video elements	Yes(from 9)	Yes	Yes	Yes
JavaScript API				
JavaScript Object Notation (JSON) parsing	Yes	Yes	Yes	Yes
WebGL	No	Yes	Partial	Partial

The DOM

Working with libraries that act at the level of the structural elements of an HTML page, we cannot avoid talking about the DOM. I will be referring to this concept often, as it is the basic structure underlying every web page. The World Wide Web Consortium (W3C) felt the need, and rightly so, to create an official standard for the representation of structured documents, so as to develop guidelines for all programming languages and platforms.The tree structure of HTML documents, as well as those of XML, follows the guidelines developed by this standard perfectly. Following is an example of an HTML document:

```
<HTML>
    <HEAD>
      <TITLE>A title</TITLE>
    </HEAD>
    <BODY>
      Hello
      <BR>
    </BODY>
</HTML>
```

The DOM tree of this document can be represented as shown in Figure 1-11.

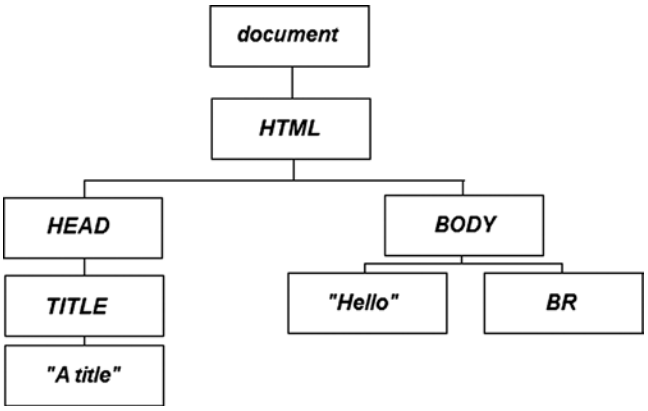


Figure 1-11. *An example of tree structure of the DOM*

But, the DOM standard is not limited to developing guidelines on how the DOM should be structured in a document; the standard also defines a number of features designed to act on the elements that compose a document. Thus, any action pertaining to a document (creating, inserting, deleting) should follow the DOM standard. As a result, regardless of the programming language that you are using or the platform on which you are working, you will always find the same functionality expressed by this standard. Often, the term *DOM* also applies to the API, which manages all the elements of a web page.

All this is important, because anyone choosing to read this book is interested in developing charts that not only use the DOM, but that are also part of it and whose every aspect can be inspected and manipulated in JavaScript. Throughout the book, you will learn how to make the best use of jQuery, jqPlot, and Highcharts (jQuery extentions) as well as D3 libraries. With these JavaScript libraries, you can access every chart element, such as changing the color and position of objects.

Developing in JavaScript

Although it is likely that most people who have chosen to read this book already have a good knowledge of JavaScript, this may not in fact be the case. For this reason, I have structured the book in a practical way, giving step-by-step examples and providing all the code that must be written in the examples. As such, this book offers newcomers an opportunity to study the language and those who have not used it for some time a chance to refresh their memories.

To start working with the JavaScript libraries that you will be using to develop your charts, it is necessary to prepare a development environment. It is true that to develop JavaScript code, you could simply use a text editor, such as Notepad (or, even better, Notepad++), but developers generally prefer to use specialized applications, usually called integrated development enviroments (IDEs), to develop code. As well as providing a text editor with differentiated colors corresponding to the keywords used in the code, such applications also contain a set of tools designed to facilitate the work. These applications can check if there are any errors in the code, supply debugging tools, make it easy to manage files, and assist in deployment on the server, among many other operations.

Nowadays, there are many JavaScript IDEs on the network, but some of the most prominent are **Aptana Studio** (see Figure 1-12); **Eclipse Web Developer**, with the JavaScript test driver (JSTD) plug-in installed; and **NetBeans**. These editors also allow you to develop Hypertext Preprocessor (PHP), CSS, and HTML (for information on how to use the Aptana Studio IDE to set up a workspace in which to implement the code for this book, see Appendix A, or use the source code accompanying the book directly; you can find the code samples in the Source Code/Download area of the Apress web site [www.apress.com/9781430262893]).

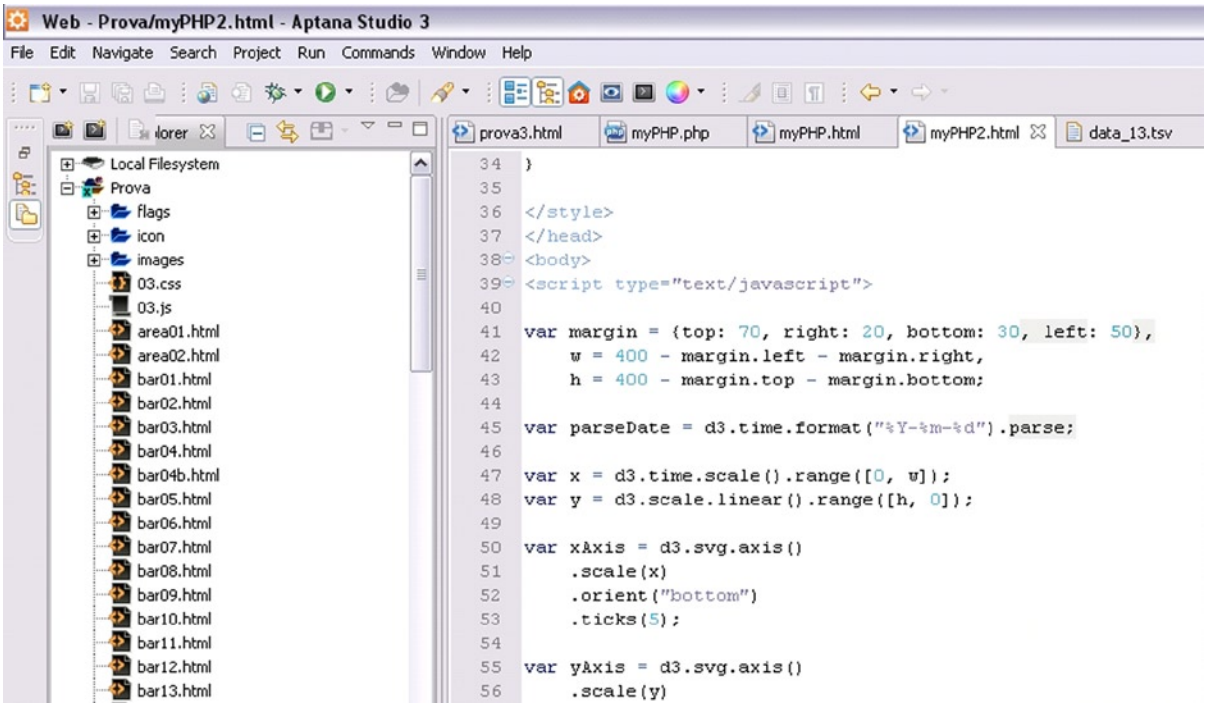


Figure 1-12. *The Aptana Studio 3 IDE*

For those who prefer not to install too many applications on their computer, there are online JavaScript IDEs. These allow users to edit JavaScript code in a web page working as an IDE and to check their result directly from the same web page. Unfortunately, many of these IDEs charge a fee. However, **jsFiddle** (`http://jsfiddle.net`) is an online IDE that requires no payment and that, in addition to editing, provides code sharing and the option of adding libraries, such as jQuery and D3.(see Figure 1-13).

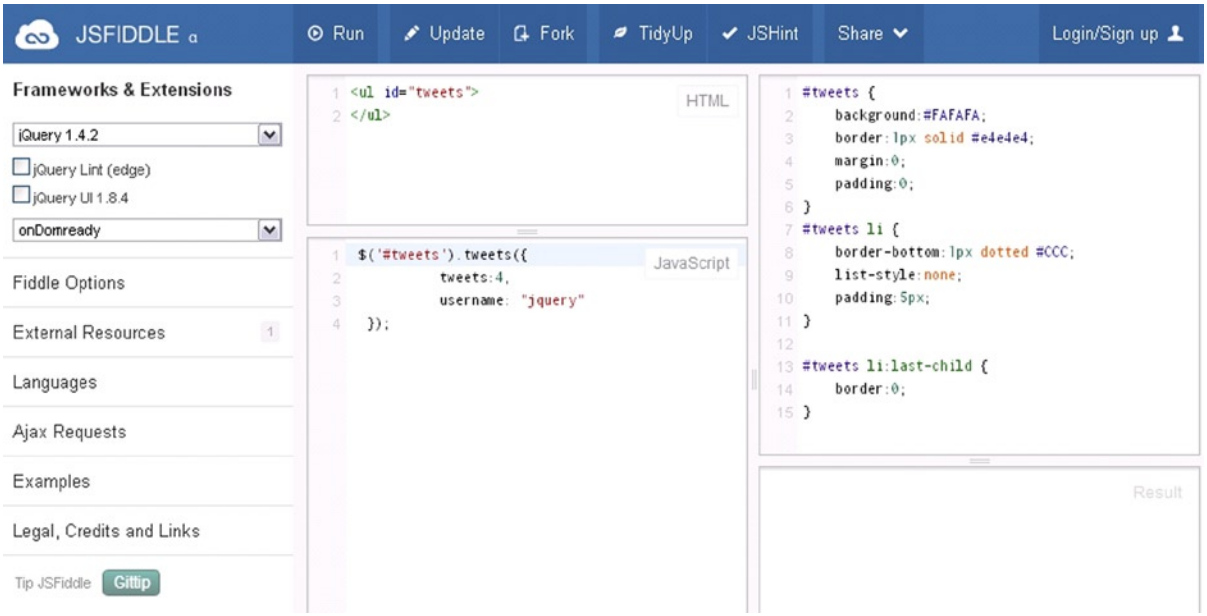


Figure 1-13. *The online IDE jsFiddle*

jsFiddle can prove very useful. As well as letting the user include many JavaScript libraries (see Figure 1-14), it offers the respective different versions released, thus allowing him or her to test any incompatibilities in real time.

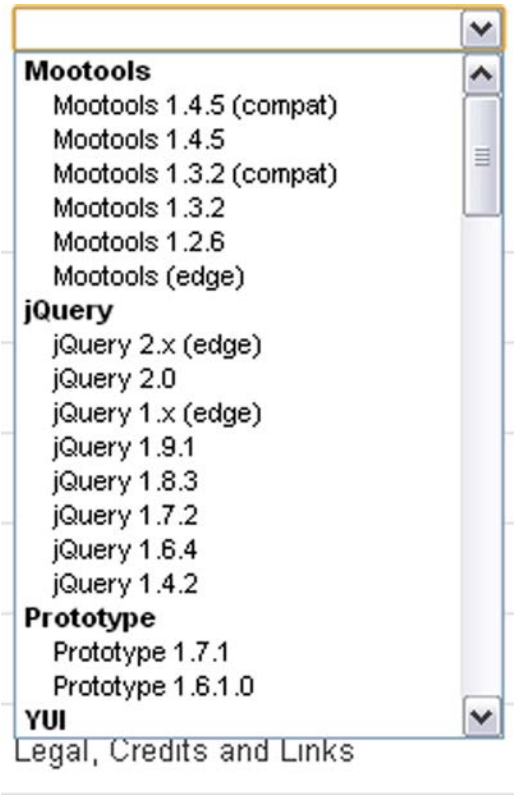


Figure 1-14. *jsFiddle offers a set of the most popular JavaScript libraries*

Running and Debugging JavaScript

JavaScript, if we want to define it in a client–server framework, is a completely client-side programming language. It is not subject to compilation, and parts of the code, apart from HTML documents, can be found in many other types of files that are specific to other languages, such as .JSP or .PHP.

These code fragments pass unaffected through the application servers without ever being processed. Only the browser is responsible for running the JavaScript code. Therefore, JavaScript code is run only when a web page is downloaded or afterward, in response to an event. If the JavaScript code is of a considerable size or might be useful subsequently, it can be defined externally in a .JS file; here, you will find all the JavaScript libraries and frameworks that will be mentioned throughout this text. Regardless of its form, however, JavaScript runs directly from the browser.

So, even if you do not use a real IDE for the development and debugging of JavaScript code, you can simply insert the code in an empty HTML file and then load this file directly in a browser (Chrome, Internet Explorer, and Firefox are the most common). To distinguish it from the rest of the text on the page, you must separate the code by putting it inside the <script></scripts> tags:

```
<script type="text/javascript">
// JavaScript code
</script>
```

If the JavaScript code resides in an external file, then it will be necessary to include it in the HTML page, writing

```
<script type="text/javascript" src="library.js"></script>
```

Therefore, as long as the execution of JavaScript is not required for the purpose of installing something, you have everything you need. Who does not have a web browser on his or her operating system?

Data Types in JavaScript

As mentioned earlier, this book will neither explain the rules and syntax for the programming of good JavaScript code nor will it linger too long on programming details. Nevertheless, the code that we are going to develop is centered on charts, or rather the processing of data and how to display them. Let us start with the simplest case. The smallest building block of all data structures is the variable (when it contains a single value). In handling the types of data, JavaScript is very different from other programming languages. , You do not have to specify the type of value (int, string, float, boolean, and so on) when you want to store JavaScript in a variable; you need only define it with the `var` keyword.

In Java or C a variable containing an integer value is declared differently from one containing a text:

```
int value = 3;
String text = "This is a string value";
```

In JavaScript the type of stored value does not matter. Everything is declared with `var`, so the same declarations are

```
var value = 3;
var text = "This is a string value";
```

Thus, in JavaScript we can think of variables as containers for storing any type of value.

For the sake of simplicity, here the variables are seen as containers of a single value, thus representing the simplest data structure. Actually, however, variables may also contain types of data that are more complex: arrays and objects.

■ **Note** The use of variables in JavaScript is actually a bit more complicated. You can also use a variable without ever declaring it with the `var` keyword. The `var` keyword will declare the variable within the current scope. If `var` is missing, JavaScript will search for a variable with the same name declared at an upper level of scope. If JavaScript does not find this variable, a new one is declared; otherwise, JavaScript will use the values in the variable found. As a result, an incorrect use of variables can sometimes lead to errors that are difficult to detect.

Arrays

An array is a sequence of values separated by a comma and enclosed in square brackets [ ]:

```
var array = [ 1, 6, 3, 8, 2, 4 ];
```

Arrays are the simplest and most widely used data structure in JavaScript, so you should become very familiar with them. It is possible to access any value in an array by specifying its index (position in the array) in the brackets, immediately following the name of the array. In JavaScript the indexes start from 0:

```
array[3]  //returns 8
```

Arrays can contain any type of data, not just integers:

```
var fruits = [ "banana", "apple", "peach" ];
```

There are a many functions that can help us handle this kind of object. Because of its usefulness, I will be using this object frequently throughout the book, and it therefore seems proper to give it a quick look.

It is possible to know the number of the values in an array by writing

```
fruits.length  //returns 3
```

Or, if you know the values, you can get the corresponding index with

```
fruits.indexOf("apple") //returns 1
```

Moreover, there is a set of functions that allow us to add and remove items in an array. **push( )** and **pop( )** functions add and remove the last element in an array, whereas **shift( )** and **unshift( )** functions add and remove the first element in an array:

```
fruits.push("strawberry");
// Now the array is [ "banana", "apple", "peach", "strawberry" ];
fruits.pop(); //returns "strawberry"
// Now the array is [ "banana", "apple", "peach"];
fruits.unshift("orange", "pear");
// Now the array is ["orange", "pear", "banana", "apple", "peach"];
fruits.shift();   //returns "orange"
// Now the array is ["pear", "banana", "apple", "peach"];
```

Sometimes, it is necessary to make a loop through every value in an array in order to perform some action with it. An approach that is widely used in other programming languages is the use of the function `for()`. For example, to calculate the sum of the values in an array, you would write

```
var sum = 0;
for (var i = 0; i < array.length; i++) {
    sum += array[i];
}
```

But, in JavaScript it is more common to use the `forEach()` function, where d assumes the values in the array, one by one, following the sequence:

```
var sum = 0;
array.forEach(function(d) {
     sum += d;
});
```

Objects

Arrays are useful for simple lists of values, but if you want structured data, you need to define an object. An object is a custom data structure whose properties and values you define. You can define an object by enclosing its properties between two curly brackets { }; every property is defined by a name followed by a colon (:) and the assigned value, and commas separate each property/value pair:

```
var animal = {
    species: "lion",
    class: "mammalia",
    order: "carnivora",
    extinct: false,
    number: 123456
};
```

In JavaScript code, dot notation refers to each value, specifying the name of the property:

```
animal.species      //Returns "lion"
```

Now that you have learned about both objects and arrays, you can see how it is possible to combine them in order to get more complex data structures in JavaScript. You can create arrays of objects or objects of arrays, or even objects of objects. Square brackets are used to indicate an array, curly brackets, an object. For example, let us define an array of objects in this way:

```
var animals = [
    {
        species: "lion",
        class: "mammalia",
        order: "carnivora",
        extinct: false,
        number: 123456
    },
    {
        species: "gorilla",
        class: "mammalia",
        order: "primates",
        extinct: false,
        number: 555234
    },
    {
        species: "octopus",
        class: "cephalopoda",
        order: "octopoda",
        extinct: false,
        number: 333421
    }
];
```

To get the values of these data structures, you need to use both the square brackets with the index and the name of the property:

```
animals[0].extinct  //return false
animals[2].species //return "octopus"
```

Firebug and DevTools

To debug, if you are using an IDE, you can easily make use of the various debugging tools that are included with it. If you do not have access to an IDE, however, you can still avail yourself of external tools. Think of the browser as a development environment, where debugging tools can be integrated through plug-ins that are downloadable from Internet. There are many tools currently available on the Internet, but the one I want to propose is Firebug, a web development tool for those who prefer to use the browser Mozilla Firefox. Firebug is an add-in that integrates seamlessly into the Firefox browser, as demonstrated in Figure 1-15.

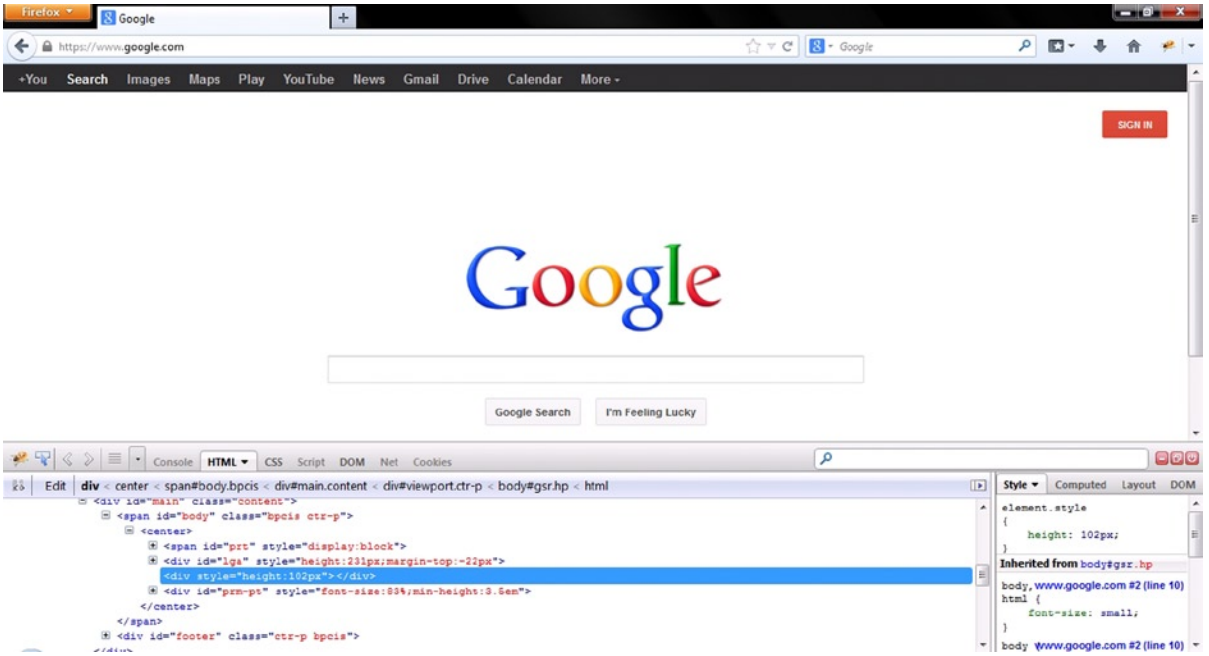


Figure 1-15. *Firebug is an extention of Mozilla Firefox and is fully integrated into the browser*

Firebug will prove a very useful tool throughout, especially when using use the jQuery and D3 libraries, which require that the structure of the DOM always be kept under control. This tool will allow you to monitor the structure of the DOM directly.

For those who prefer to use Google Chrome, however, there is DevTools, which is already integrated into the browser (see Figure 1-16). To access this tool, simply click the button at the top-right corner of the browser.

Next, select Tools ➤ Developer Tools, or simply right-click any page element, and then select Inspect element in the context menu.

With these two tools, you can easily inspect not only each element of the DOM—its attributes and their values—but also the CSS styles applied to them. You can also input changes to these values to observe the effects

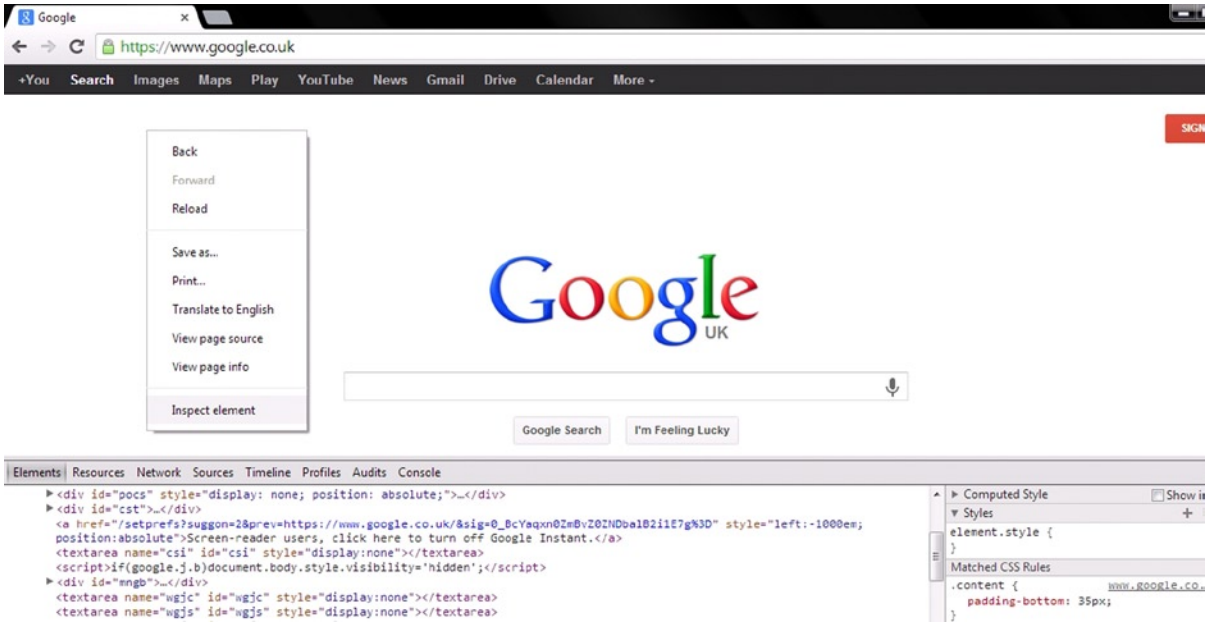


Figure 1-16. *With DevTools it is possible to monitor a lot of information about your web page*

in real time without having to modify the code on file and save it every time. Firebug and DevTools also include various tools for monitoring the performance of the page, for both rendering and networking.

With DevTools, particular attention should be paid to the use of the console as well. Through it, you can access diagnostic information, using methods such as `console.log()`. This method is frequently used to display the values of many variables through the console, passing the name of the variable as an argument, with the addition of text as an indication:

```
var x = 3;
console.log("The value of x is " + x); // The value of x is 3
```

It is also possible to enter commands and perform interactions with the document, using methods such as `$()` or `profile()`.For further information on these methods, see the documentation regarding the Console API (https://developers.google.com/chrome-developer-tools/docs/console-api) and the Command Line API (https://developers.google.com/chrome-developer-tools/docs/commandline-api).

JSON

JSON is a specific syntax for organizing data as JavaScript objects. This format is generally used in browser-based code, especially in JavaScript. JSON represents a valid alternative to XML for organizing data. Both are independent from the programming language, but JSON is faster and easier to parse with JavaScript than XML, which is a full-markup language. Moreover, jqPlot and D3 work well with JSON. Its structure follows perfectly the rules that we have seen for objects and arrays defined in JavaScript:

```
var company = {
    "name": "Elusive Dinamics",
    "location": "France",
    "sites": 2,
    "employees": 234,
    "updated": true
};
```

Summary

This first chapter has introduced you to many of the fundamental concepts that will accompany you throughout the book. First, you examined the most common types of charts and the elements that compose them. You also took a quick look at many of the technical aspects you need to know when setting about developing a chart on the Web. Finally, you briefly explored the types of data used in the JavaScript examples in this book.

I mentioned that the majority of your work will be done by specialized JavaScript libraries. In the next chapter, you will learn about the **jQuery** library. This library will provide you with a whole range of tools that act directly, at the level of the DOM. Later in the book, you will find that knowledge of this library is vital: many of the graphics libraries (including jqPlot and Highcharts) are built on it.

CHAPTER 2

jQuery Basics

In the previous chapter, you learned about the DOM tree and saw how HTML documents are composed of many elements that can be created, modified, and deleted from the initial context. These operations are performed by the browser via JavaScript commands that, as discussed previously, can be executed either at the time of page loading or as a result of events that follow. A JavaScript library that manages all these operations in a simple and well-established manner has been developed for this purpose. This library is jQuery, and it is completely open source. It was created in 2006 by Jon Resig and continues to be improved on by a team of developers. Because of its usefulness, compared with classic JavaScript, and its ability to manipulate DOM elements, jQuery is currently the most widely used JavaScript library and constitutes a point of reference for all web developers.

Any developer who plans to include the jQuery library in a web page will soon discover the truth of the now well-known motto that accompanies thisUI libraries: "Write less, do more." In the spirit of this slogan, jQuery has introduced three new concepts in the development of JavaScript code—concepts you need to keep in mind when using the methods provided by thisUI libraries:

- Choosing elements of the HTML page (selections) on which to apply jQuery methods through Cascading Style Sheets (CSS) selectors
- Building chains of jQuery methods, applicable in sequence on a same selection
- Making implicit iterations, using jQuery wrappers

In this chapter, after seeing how to include the jQuery library in the Web pages that you will develop, the concept of the "selection" will be introduced. Selections are the base of the jQuery library and it will be important to understand them and how to implement them, as they will be discussed throughout the book. Through a series of small examples and using the technique of chaining methods, you will browse a range of functions provided by the jQuery library that will allow you to manipulate the selections in order to create, hide, and change the various DOM elements and their content.In the last part of this chapter, I will introduce the **jQuery user interface library** (**jQuery UI**), illustrating some of its most common widgets . You will learn their basic features and discover how to incorporate them within a web page.

The aim of this chapter is to provide a quick view of jQuery—its functionality and basic concepts. A detailed description of each of its methods is beyond the scope of this book. In all the examples in this text, these methods will be explained contextually. However, you may also want to consult the documentation on the official jQuery web site (`http://jquery.com/`).

Including the jQuery Library

Now, there are two ways to include the jQuery library in your web page.

- **Local method**: Download the necessary libraries locally, and then add them to the web page.
- **CDN method**: Set the link directly to the sites that provide these JavaScript libraries.

The sites offering these libraries are known as content delivery networks (CDNs). A CDN is a large system of servers that provide content with high availability and high performance to end users. When a user attempts to access a file, the CDN picks the server nearest to the user. One of the most used CDNs is Google Hosted Libraries. This service supplies web applications with reliable access to many of the most popular open source JavaScript libraries, such as jQuery.

To load the libraries from this service, you need to copy the following HTML snippet and paste it directly into your web page:

```
<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js">
</script>
```

Another CDN site from which to obtain any version of the jQuery library is the official CDN site of jQuery itself: **code.jquery.com**. If you prefer to use this site's URL, you need to copy and paste this HTML snippet:

```
<script src="http://code.jquery.com/jquery-1.9.1.min.js"></script>
```

■ **Note** All the examples in this chapter use version 1.9.1 of the jQuery library.

If you choose to follow the local option instead, you need to copy and paste the relative path of the jQuery library. This path will vary, depending on where the library is situated on the web server or on your PC. It is good practice to create an appropriate local directory for loading all the libraries you will need to include.

For example, you may decide to put your jQuery library in an `src` directory and the web pages that you are developing in a `charts` directory, as shown in Figure 2-1. In this case, you have to use this relative path:

```
<script src="../src/js/jquery-1.9.1.js"></script>
```

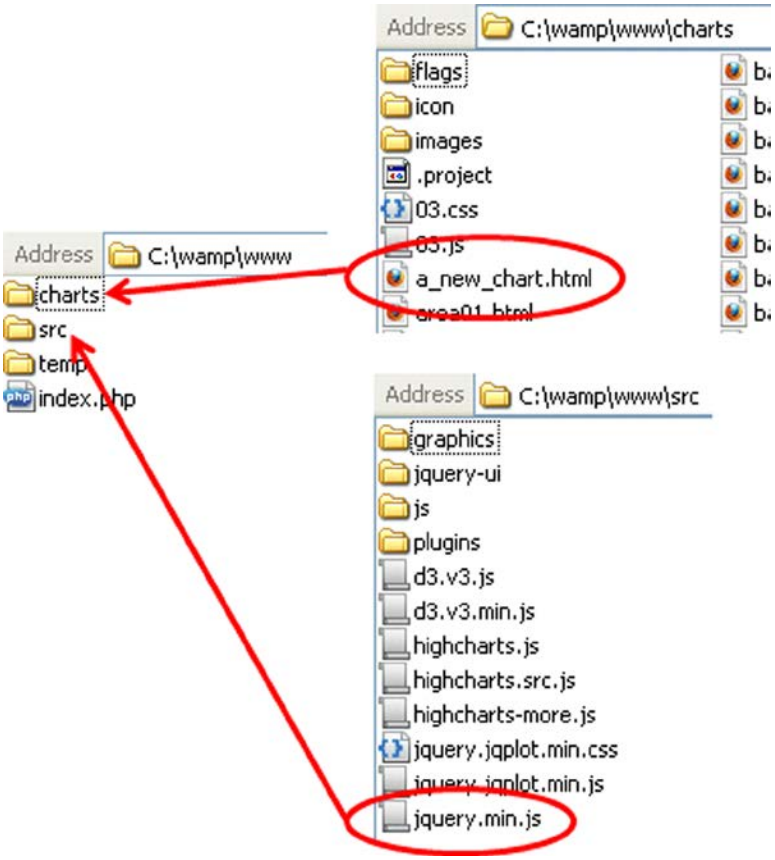


Figure 2-1. *An example of how a directory might be organized on a web server*

■ **Note** For details on how to set up on a web server or PC a workspace in which to develop the examples in this book, see Appendix A. You will also find information on the different versions of the libraries, how to download them, and how to include them in the workspace.

Selections

A selection is a group of HTML elements that have been chosen in order to be manipulated in some way. In effect, this is the main concept behind jQuery. Let us take as an example the simple HTML page in Listing 2-1:

Listing 2-1. ch2_01a.html

```
<HTML>
<HEAD>
<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js"></script>
</HEAD>
<BODY>
```

```
<div> This is the first text</div>
<div class="selected"> This is the second text</div>
<div> This is the last text</div>
</BODY>
</HTML>
```

In this page there are three elements containing three different texts. The second element in the list has been marked with the class name 'selected'. To select all three elements, you can use the selector 'div', which identifies them among all the elements on the page.

Next, you write the `jQuery()` function, with the selector passed as an argument. In this way, you will have selected the three elements and their contents. To get the text, you use the function `text()`, placing it in chain to the `jQuery()` function call and adding this line at the end of the <body> section, as shown in Listing 2-2.

Listing 2-2. ch2_01a.html

```
<script>
var text = jQuery('div').text();
console.log(text);
</script>
```

All text contained in the three elements has been assigned to the variable text. To view its contents (very useful in debugging), you can use the function `console.log()` and then, on Google Chrome, select Inspect element by right-clicking the page directly (see Figure 2-2).

Figure 2-2. *The variable text contains the text within the three elements*

Depending on the frequency with which you make your selections, you can also call this function with `$()`, as shown in Listing 2-3. We will be using this option in the examples provided in this book.

Listing 2-3. ch2_01b.html

```
<script>
var text = $('div').text();
console.log(text);
</script>
```

In contrast, if you want to select only one of the three elements, you can distinguish them by assigning a class name to each element and then apply the selector to the name of the element chosen, instead of the tag element (Listing 2-4).

Listing 2-4. ch2_01c.html

```
<script>
var text = $('.selected').text();
console.log(text);
</script>
```

In this case, the variable text contains only the text of the second element, as demonstrated in Figure 2-3.

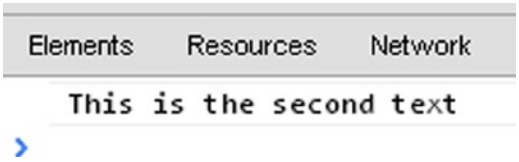


Figure 2-3. *The content of the text variable displayed by Inspect element in the Google Chrome browser*

Once you understand how to make a selection, you will discover how to manipulate any element by changing its content or attributes. You can even add other elements or remove an element from the page. In this regard, jQuery provides us with the necessary tools, thanks to the large number of methods it affords.

Chaining Methods

jQuery is designed to allow jQuery methods to be chained. Once the selection of an element or a set of elements is made, the next step is to apply a sequence of methods to it. This sequence can be written using chaining.

Using the previous example (see Listing 2-1), let us say you want to replace the text in the second element with another phrase and hide the other two elements so that they no longer appear in the web page. To do this, you are going to replace the line "This is the second text" with a new line, "This is a new text," hiding the other text at the same time. Figure 2-4 shows what appears before any change is applied.

This is the first text
This is the second text
This is the last text

Figure 2-4. *The text displayed by the web page without using the jQuery methods chain*

Now, you apply the following chain of methods:

```
$('div').hide().filter('.selected').text('This is a new text').show();
```

All three elements are included in the selection and then hidden. In the selection you chose, only the elements with the class name '`selected`' and their content are replaced with a new text. Only these last elements must be shown. So, at the end of this chain of command, the result will be

This is a new text

The Wrapper Set

When jQuery is involved, we deal with wrapper sets. In the previous example, there are three elements. You will often make selections containing several elements, but you will never need to specify a programmatic loop. Here, when you applied the `hide()` method in order to hide all three elements, you did not use a `for` or `while` construct (i.e., `$('div').each(function() {})`. Therefore, a **wrapper set** may be defined as a group of elements (selection) amenable to any manipulation, as if it were a single item.

jQuery and the DOM

jQuery is a library that principally works on the document object model (DOM) and that always refers to it for all its features. jQuery, like the DOM, treats each web page like a tree structure, in which each tag (also called element) is a node. The root of this tree is the document, which is the element that contains all the other elements of the DOM. jQuery provides a set of methods that simplify the manipulation of this kind of object, allowing you to add dynamism to your page.

The ready( ) Method

If you want to write a JavaScript code that manipulates DOM elements, the DOM needs to be loaded before you can operate on it. But, you need to operate before the browser has loaded all assets completely. To this end, jQuery provides you with the `ready()` method. This is a custom event handler that is bound to the DOM of the document object. The `ready()` method takes only one parameter: a function containing the code, which should be executed just after the DOM is loaded, but before the user can see all the assets in the browser.

```
$(document).ready( function() {
    // we write the JavaScript code here.
});
```

Traversing the DOM with Selections

You have seen how to select a group of DOM elements, using a specific CSS selector passed as an argument that identifies them. However, the potential of jQuery does not end there; starting from the position of the selection within the DOM, it is possible to traverse the DOM to get a new set of selected elements to operate on. jQuery provides us with a set of methods to apply to a selection.

Let us take as an example the simple HTML code in Listing 2-5:

Listing 2-5. ch2_03a.html

```
<HTML>
<HEAD>
<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js"></script>
</HEAD>
<BODY>
<div class="fruits">
  <div>Apple</div>
  <div>Orange</div>
  <div>Banana</div>
  <div>Strawberry</div>
</div>
</BODY>
</HTML>
```

This page will show a list of four fruits. As you have already seen, if you make a selection with '`div`' as selector, you will get a sequence of the five elements:

```
<div class="fruits">
    <div>Apple</div>
    <div>Orange</div>
```

```
    <div>Banana</div>
    <div>Strawberry</div>
</div>
<div>Apple</div>
<div>Orange</div>
<div>Banana</div>
<div>Strawberry</div>
```

You need to pay special attention to the first element. You will find the other four fruits in the selection, although these will then be repeated in successive elements. This is because the selector 'div' selects every element, along with its contents, regardless of whether an element therein will in turn be selected. It is important to take this into account whenever you want to subject this type of selection to further manipulations.

Now, if you write the snippet in Listing 2-6, you get the text in an alert dialog box, as shown in Figure 2-5. , You can see that the text in the last row includes all fruits.

Listing 2-6. ch2_03a.html

```
<script>
  var text = $('div').text();
  alert(text);
</script>
```

Figure 2-5. *The alert dialog box shows the text contained within the elements of the selection*

Often, however, you need to access a specific value of the selection directly. For example, to access the second element of the current selection directly, you can write

```
var text = $('div:eq(1)').text();
```

You have used the function `eq()` with the index of the element in the selection you wish to choose. Now, you have only this text:

Similarly, if you want to select the third element of the sequence, you can directly write

```
var text = $('div:eq(2)').text();
```

Or, if you prefer, you can make a traversing, using the `next()` method to move the selection from one element to the next:

```
var text = $('div:eq(1)').next().text();
```

You then get this alert message:

Now, let us look at an example that demonstrates the difference between the selection and the DOM structure. This sometimes causes confusion. You must remember that the `eq()` method makes a sort of subselection; `next()`, `prev()`, `parent()`, `children()`, `nextAll()`, and `prevAll()` shift the selection onto the DOM.

In fact, if you write the chain

```
var text = $('div:eq(1)').prev().text();
```

you do not get anything, because the element selected by '`div:eq(1)`' is the first on the list (but second in the selection). Therefore, if you try to shift the selection to a previous element in the DOM, you do not get anything. If you want to shift the selection to the parent element, called '`fruits`', you need to use the `parent()` method:

```
var text = $('div:eq(1)').parent().text();
```

Now, you get the parent element, which is the same as the first element of the selection. Figure 2-6 presents the result.

Figure 2-6. *The alert dialog box shows the four fruits within the first element*

Had you written the command

```
var text = $('div:eq(0)').text();
```

you would have obtained the same result.

Create and Insert New Elements

So far, you have seen that by passing an argument in the function `jQuery()` or in its alias `$()`, you obtained a selection of all the items that have that tag in the DOM or the same class name. Now, suppose you pass as an argument a tag that is not present in the HTML page. Here, you have just created a new item to add to the DOM. Moreover, this element is, for all intents and purposes, a selection and may therefore be subjected to any kind of manipulation, even if it has not yet been physically added to the web page. By adding some specific jQuery methods at the end of the method chain, you will decide where to insert the newly created element.

For instance, as shown in the previous example, by writing the snippet

```
$('<div>Lemon</div>').appendTo('div:eq(2)');
```

you create a new element in the list of fruits. Then, you append it after the third element of the selection (the second element of the list). Figure 2-7 shows how the list in the web page appears after the change is applied .

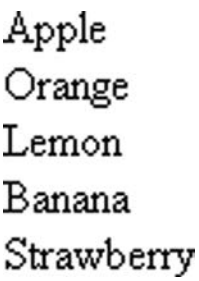


Figure 2-7. *The list can be increased dynamically, adding new elements*

There are many methods that specify where and how to insert the elements just created: `prepend()`, `after()`, `before()`, `append()`, `appendTo()`, `prependTo()`, `insertAfter()`, `insertBefore()`, `wrap()`, `wrapAll()`, `wrapInner()`, and so on.

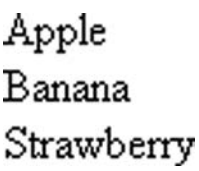
For more details on the use of these functions, the reader is advised to consult the documentation on the official jQuery web site (`http://jquery.com/`).

Remove, Hide, and Replace Elements

Another set of very useful jQuery methods includes those methods that allow us to eliminate static elements from the page (from the DOM) or at least to hide them. Sometimes, these methods can be useful even for replacing one element with another.

To remove the "Orange" fruit from the list, simply write

```
$('div:eq(2)').remove();
```



If you want to hide it, you write

```
$('div:eq(2)').hide();
  ...
$('div:eq(2)').show();
```

In this case, however, further on in the code, it will be possible to show "Orange" again.

If you use `remove()` instead (see Listing 2-7), the element corresponding to the selector '`div:eq(2)`' changes, and it would not be possible to recover the removed element.

Listing 2-7. ch2_04c.html

```
$('div:eq(2)').remove();
var text = $('div:eq(2)').text();
alert(text); //returns 'Banana'
```

Finally, if you want to replace “Orange” with “Pineapple,” you can do so with the `replaceWith()` method, as follows:

```
$('div:eq(2)').replaceWith('<div>Pineapple<div>');
```

Now, you have a new list of fruits, as demonstrated in Figure 2-8.

Figure 2-8. *The list can be dynamically reduced by removing some of its elements*

jQuery UI: Widgets

Along with the jQuery library, there is another library that can help you integrate your web page with interactive and graphic objects: the jQuery UI. This library provides a whole range of tools, such as widgets, themes, effects, and interactions, that enrich web pages, turning them into highly interactive web applications. For our purposes, widgets are of particular interest. These small graphic applications can prove a valuable tool that, when added to your web pages, makes your charts even more interactive. Widgets facilitate interaction the program beneath the web page and very often are real mini-applications. In their simplest forms, widgets appear as tables, accordions, combo boxes, or even buttons.

As with the jQuery library, you will need to include the plug-in file in the web page if you want to integrate its widgets. You must also include the CSS file representing the theme. This can be done through the Google Hosted Libraries service:

```
<link rel="stylesheet" href="http://ajax.googleapis.com/ajax/libs/jqueryui/1.10.3/themes/smoothness/
jquery-ui.css" />
<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js">
</script>
<script src="http://ajax.googleapis.com/ajax/libs/jqueryui/1.10.3/jquery-ui.min.js">
</script>
```

You can also download from the official CDN jQuery site:

```
<link rel="stylesheet" href="http://code.jquery.com/ui/1.10.3/themes/smoothness/jquery-ui.css" />
<script src="http://code.jquery.com/jquery-1.9.1.min.js"></script>
<script src="http://code.jquery.com/ui/1.10.3/jquery-ui.min.js"></script>
```

If you prefer to download the libraries locally or to use the workspace in the source code accompanying this book (see Appendix A), you can refer to the libraries as follows:

```
<link rel="stylesheet" href="../src/css/smoothness/jquery-ui-1.10.3.custom.min.css" />
<script src="../src/js/jquery-1.9.1.js"></script>
<script src="../src/js/jquery-ui-1.10.3.custom.min.js"></script>
```

■ **Note** The theme for the jQuery UI widgets used in this book is **smoothness**. The list of available themes is vast and covers many combinations of colors and shapes. This well-stocked list is available on ThemeRoller (http://jqueryui.com/themeroller). ThemeRoller is a page on the official jQuery web site that allows you to preview widgets and to then download your favorite theme from those available.

On visiting the official jQuery UI web site (http://jqueryui.com/), you will notice that the widgets provided by this library are numerous. Here, I will discuss only the most common examples, especially those that are most likely to be integrated into a page containing charts.

As you will see throughout this book, some of these widgets will be used as containers, exploiting their particular capabilities, such as resizing and encapsulation, including these:

- Accordions
- Tabs

Other widgets will be used to replace the simple controls that HTML offers, as the former are much more advanced and rich in functionality, including the following:

- Buttons
- Combo boxes
- Menu
- Sliders

Still other widgets will also perform the function of indicators. With these, you will see how to integrate a particular widget class:

- Progress bars

Accordion

An accordion widget is a set of collapsible panels that enable the web page to show a large amount of information in a compact space. Each panel can hold a thematic area or, as you will see in later chapters, different types of charts. The content is revealed by clicking the tab for each panel, allowing the user to move from one panel to another without changing the page. The panels of the accordion expand and contract, according to the choice of the user, such that only one panel shows its content at any given time.

The HTML structure you need to write in order to obtain an accordion widget in the page is composed of an outer tag containing all the panels. Each panel in turn is specified by a heading placed between two <h3> tags and a pair, with the content in between. Listing 2-8 represents a simple accordion with four panels.

Listing 2-8. ch2_05.html

```
<div id="accordion">
  <h3>First header</h3>
  <div>First content panel</div>
  <h3>Second header</h3>
  <div>Second content panel</div>
  <h3>Third header</h3>
  <div>Third content panel</div>
```

```
  <h3>Fourth header</h3>
  <div>Fourth content panel</div>
</div>
```

In JavaScript code, you need to add the snippet in Listing 2-9 in order to obtain an accordion widget.

Listing 2-9. ch2_05.html

```
$(function() {
    $( "#accordion" ).accordion();
});
```

Figure 2-9 illustrates our accordion.

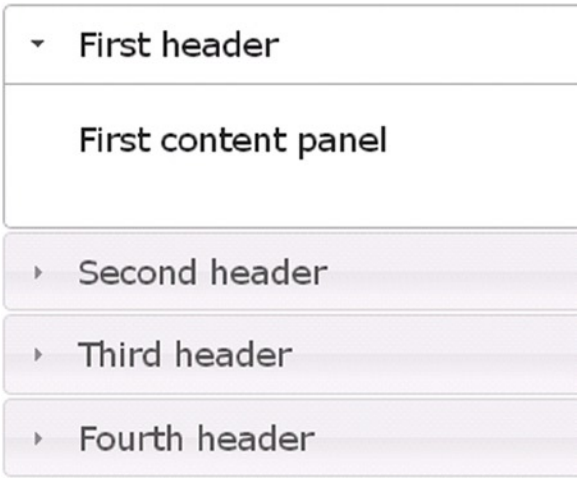


Figure 2-9. *An accordion consists of collapsible panels suitable for containing information in a limited amount of space*

But, that is not enough. It would be better if you could control the style of the accordion. This can be accomplished by adding the code given in Listing 2-10.

Listing 2-10. ch2_05.html

```
<style type="text/css">
  .ui-accordion {
      width: 690px;
      margin: 2em auto;
  }
  .ui-accordion-header {
      font-size: 15px;
      font-weight: bold;
  }
  .ui-accordion-content {
      font-size: 12px;
  }
</style>
```

The result is shown in Figure 2-10.

Figure 2-10. *By modifying the CSS style properties, you can change the accordion's appearance as you like*

Tab

A widget that is very similar to the accordion in its functionality is the panel with tabs. Here, each panel is unique, but there are several tabs at the top, identified by different headings. Nonetheless, this widget affords the possibility to show a large amount of information in a limited space, and the user can choose to view the content of only one tab at a time. More significant is the loss of the vertical expansion of panels.

The HTML structure you need to write in order to obtain a tab widget in the web page is slightly more complex than the previous one. The headings are given in an unordered list <ul>, in which each item <li> must be referenced to an anchor tag <a>. The contents of every tab are enclosed in a pair, with an `id` attribute corresponding to the references in the anchor tags (see Listing 2-11).

Listing 2-11. ch2_06.html

```
<div id="tabs">
  <ul>
    <li><a href="#tabs-1">First header</a></li>
    <li><a href="#tabs-2">Second header</a></li>
    <li><a href="#tabs-3">Third header</a></li>
    <li><a href="#tabs-4">Fourth header</a></li>
  </ul>
  <div id="tabs-1">
    <p>First tab panel</p>
  </div>
  <div id="tabs-2">
    <p>Second tab panel</p>
  </div>
   <div id="tabs-3">
    <p>Third tab panel</p>
  </div>
  <div id="tabs-4">
    <p>Fourth tab panel</p>
  </div>
</div>
```

In JavaScript code, you need to specify the tab widget, as shown in Listing 2-12.

Listing 2-12. ch2_06.html

```
$(function() {
    $( "#tabs" ).tabs();
});
```

The CSS style classes must also be defined, as shown in Listing 2-13.

Listing 2-13. ch2_06.html

```
<style type="text/css">
    .ui-tabs {
      width: 690px;
      margin: 2em auto;
    }
    .ui-tabs-header {
      font-size: 15px;
      font-weight: bold;
    }
    .ui-tabs-panel {
      font-size: 12px;
    }
</style>
```

When the procedure is complete, you will get the widgets illustrated in Figure 2-11.

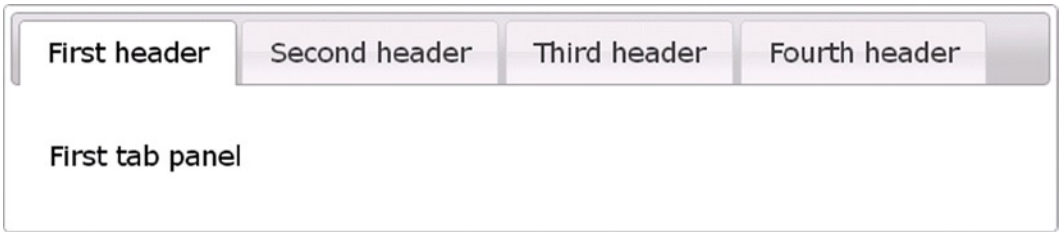


Figure 2-11. *The tab widget consists of multiple panels that occupy the same area*

Button

Among all the widgets available, the button remains the most commonly used. Previously, there were two ways to insert a button in a web page. The first was the classic method, with the tag `<input type="button"/>`. A more modern approach was the <button> tag. But, thanks to jQuery, there is another kind of button that we have not yet considered. We can create an anchor tag <a> as a button, calling it "anchor button." When the user clicks it on the page, the browser loads the corresponding link. How to insert in a blank web page all three of the examples described is shown in Listing 2-14.

Listing 2-14. ch2_07.html

```
<button>A button element</button>
<input type="submit" value="A submit button" />
<a href="#">An anchor</a>
```

Without further specification or JavaScript code, when you load the page, you see the buttons presented in Figure 2-12.

Figure 2-12. *The web page shows three types of buttons: a simple button element, a submit button, and an anchor button*

To refer to them by using a JavaScript function, write the snippet provided in Listing 2-15.

Listing 2-15. ch2_07.html

```
$(function() {
    $( "input[type=submit], a, button" )
      .button()
      .click(function( event ) {
         event.preventDefault();
      });
});
```

In this way, you will get a more presentable set of buttons, as demonstrated in Figure 2-13.

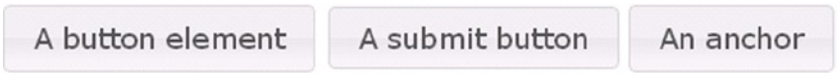


Figure 2-13. *The three types of buttons are now represented by the jQuery UI button widgets*

You can enrich your buttons by adding icons. jQuery UI offers a huge set of icons, but you may also use larger, personal icons. Listing 2-16 shows how to write the four buttons into your web page:

Listing 2-16. ch2_08.html

```
<button>Button with icon only</button>
<button>Button with custom icon on the left</button>
<button>Button with two icons</button>
<button>Button with two icons and no text</button>
```

You have added four buttons to highlight four possible cases: a button with only an icon; a button with text and an icon on the left side; a button with text and an icon on each side; and a button with two icons and no text (see Figure 2-14). Looking at the HTML code, you can see that actually all four buttons have text inside, but this feature can be disabled in order to get a button without text. Listing 2-17 illustrates the assignment of icons to the various buttons, with the icon name being assigned to the `primary` and `secondary` (optional) attributes. Furthermore, by setting the `text` attribute to '`false`', you can obtain a button without text.

Listing 2-17. ch2_08.html

```
$(function() {
    $( "button:first" ).button({
        icons: {
            primary: "ui-icon-locked"
        },
        text: false
    }).next().button({
        icons: {
            primary: "ui-icon-italy"
        }
    }).next().button({
        icons: {
            primary: "ui-icon-gear",
            secondary: "ui-icon-triangle-1-s"
        }
    }).next().button({
        icons: {
            primary: "ui-icon-gear",
            secondary: "ui-icon-triangle-1-s"
        },
        text: false
    });
});
```

To insert customized icons, you need to define their address as a CSS file, using the function `url()`, as demonstrated in Listing 2-18.

Listing 2-18. ch2_08.html

```
<style>
.ui-button .ui-icon-italy {
    background-image: url("icon/exit24x24.png");
    width: 24px;
    height: 24px;
}
</style>
```

Figure 2-14 shows the set of buttons you have just created.

Figure 2-14. *Each button can be easily enriched with icons*

Combo Box

The combo box is another widely used control in web pages and in many applications. A combo box is an editable drop down menu, from which the user can select an entry. To insert a combo box in your page, you need to define a specific structure of elements, as shown in Listing 2-19.

Listing 2-19. ch2_09.html

```
<div class="ui-widget">
    <label>Select your destination:</label>
    <select id="combobox">
        <option value="">Select one...</option>
        <option value="Amsterdam">Amsterdam</option>
        <option value="London">London</option>
        <option value="Rome">Rome</option>
    </select>
</div>
```

Next, you need to refer this structure using the JavaScript code, first choosing the elements with `$()` and then activating the structure as a jQuery combo box widget:

```
$(function() {
     $( "#combobox" ).combobox();
});
```

Let us add a pinch of CSS style:

```
<style>
    .ui-widget {
       font-size: 18px;
    }
</style>
```

Figure 2-15 illustrates the combo box widget, which represents the starting point for a whole series of functionalities for enabling the capture of events.

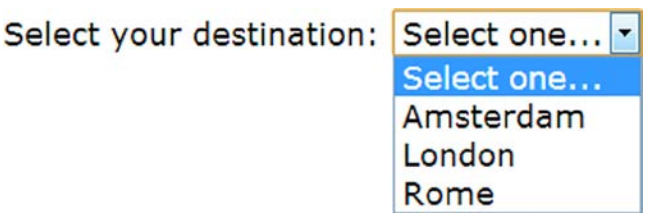


Figure 2-15. *A combo box is a drop-down menu allowing the user to make a choise among various options*

Menu

Having just considered the combo box, you cannot overlook the possibility of including an interactive menu on your home page. With such a menu, the user can make a series of choices, such as selecting options on how to represent a chart.

In HTML an unordered list is defined as <ul>, and a list of items, as <li>. If you want to add a submenu as an item, you only need to insert an embedded unordered list <ul>. To illustrate how to build a menu, let us take a look at Listing 2-20.

Listing 2-20. ch2_10.html

```
<ul id="menu">
    <li class="ui-state-disabled"><a href="#">Advanced</a></li>
    <li><a href="#">Filter</a></li>
    <li>
        <a href="#">Zoom</a>
        <ul>
            <li><a href="#">10%</a></li>
            <li><a href="#">25%</a></li>
            <li><a href="#">50%</a></li>
            <li><a href="#">100%</a></li>
        </ul>
    </li>
</ul>
```

As with the preceding widgets, you must activate the menu by adding the following function:

```
$(function() {
    $( "#menu" ).menu();
});
```

You also need to include the CSS style settings:

```
<style>
    .ui-menu {
        width: 150px;
    }
</style>
```

Now, you have a menu on the page, as shown in Figure 2-16.

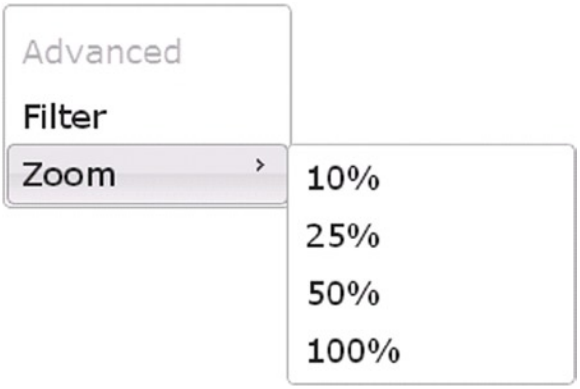


Figure 2-16. *A drop-down menu lets you categorize different options*

Slider

When you begin to develop various types of charts, you will find that several parameters have to be set each time. These parameters may be modified in real time by the user by means of sliders. These sliders enable the user to change parameters within a certain range.

As with many of the other widgets, first you add the element to represent the slider in the web page.

```
<div id="slider"></div>
```

Then, as always, you activate the widget with a JavaScript function, specifying the widget's attributes within. For example, to specify the default position of the slider handle, you set the `value` attribute to a percentage value ranging from 0 to 100. Similarly, the orientation can be set by assigning the string '`horizontal`' or '`vertical`' to the `orientation` attribute. For the `range` attribute, you can indicate if the range (the shaded area of the slider track) covered by the slider should start from the '`min`' value or the '`max`' value (see Figure 2-19). Thus, if you were to set the range attribute to '`min`', the range would extend from the minimum value to the slider handle. The `animation` attribute is another setting to consider. The slider widget has animation built in: when the user clicks the slider track, the handle moves from its current position to reach the clicked point; this can be done slowly or quickly. You can choose how fast the handle moves by setting the `animation` attribute to '`fast`' or '`slow`'. The attributes '`true`' and '`false`' indicate if the animation is enabled or disabled (see Listing 2-21).

Listing 2-21. ch2_12.html

```
$(function() {
    $( "#slider" ).slider({
        value: 60,
        orientation: 'horizontal',
        range: 'min',
        animate: 'slow'
    });
});
```

Once you have defined the fundamental attributes of the slider, you have to decide its size (and that of the handle) and add CSS style settings. When you define the length and width of the slider, you need to take into account the orientation you have chosen, setting the `height` and `width` attributes accordingly. In this case, we want to represent a slider horizontally; thus, the `width` attribute will be far greater than the `height` attribute (see Listing 2-22).

Listing 2-22. ch2_12.html

```
<style>
    .ui-slider {
       width: 400px;
       height: 10px;
    }
    .ui-slider .ui-slider-handle {
       width: 12px;
       height: 20px;
    }
</style>
```

If you load the web page in a browser, you can see the slider (see Figure 2-17).

Figure 2-17. *A slider is a widget that allows you to select a numeric value in a range*

Sometimes, you will need to use multiple sliders; you will need to organize these horizontally (you can find a similar structure, e.g., in the equalizer of a stereo). When specifying several sliders, it is not necessary to define multiple elements: all that is required is a single element, with `"eq"` as its `id`, to mark it. Then, within this element, you define each slider as a pair containing its default value (i.e., where the respective handles appear on the slider tracks), as demonstrated in Listing 2-23.

Listing 2-23. ch2_13.html

```
<div id="eq">
    <span>88</span>
    <span>77</span>
    <span>55</span>
    <span>33</span>
    <span>40</span>
    <span>45</span>
    <span>70</span>
</div>
```

Now, you must implement a JavaScript function, this time one that is slightly more complex. First, using the `$("#eq > span")` selector, you make a selection on the seven elements. Then, with the `parseInt()` function, you assign all the values contained in pairs to the corresponding `value` attributes in order that the handles be in the positions shown in Figure 2-18 (see Listing 2-24).

Listing 2-24. ch2_13.html

```
$(function() {
    $( '#eq > span' ).each(function() {
        // read initial values from markup and remove that
        var value = parseInt( $( this ).text(), 10 );
        $( this ).empty().slider({
            value: value,
            range: 'min',
            animate: 'slow',
            orientation: 'vertical'
        });
    });
});
```

Even for this equalizer-like structure, it is necessary to add some CSS style settings, such as the margins between the different sliders (see Listing 2-25).

Listing 2-25. ch2_13.html

```
<style>
    #eq span {
      height:180px;
      float:left;
```

```
      margin:15px;
      width:10px;
    }
</style>
```

In the end, you get the bars illustrated in Figure 2-18.

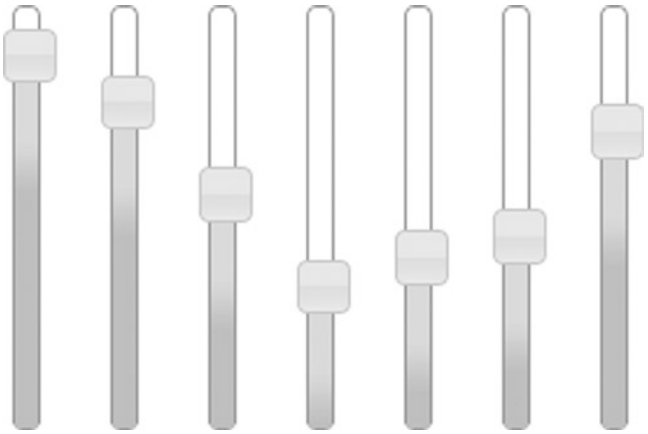


Figure 2-18. *The sliders can also be grouped in series to achieve more complex controls (e.g., an equalizer)*

Progress Bar

When you are developing complex operations, the system may require a long time to finish its tasks. While the user is on hold, to prevent the system from appearing to be locked, it is usual to represent the percentage of the process completed with a progress bar. Defining a progress bar is very simple:

```
<div id="progressbar"></div>
```

You must also write the corresponding function in JavaScript in order to activate the progress bar, as shown in Listing 2-26.

Listing 2-26. ch2_11a.html

```
$(function() {
    $( "#progressbar" ).progressbar({
        value: 37
    });
});
```

Next, you add the CSS style settings, as illustrated in Listing 2-27.

Listing 2-27. ch2_11a.html

```
<style>
    .ui-progressbar {
       height: 20px;
       width: 600px;
    }
</style>
```

But, what you get is not the desired result; you get a static progress bar, fixed at the 37 percent mark (see Figure 2-19).

Figure 2-19. *With a progress bar, you can display the status of a process*

To obtain a fully functional progress bar, you need to set its attribute `value` with a counter value directly connected to the underlying iteration of the process. Furthermore, if you want to increase the dynamism of the progress bar, you can use an animated graphics interchange format (GIF) image as its background. Listing 2-28 displays the addition of CSS style properties to the progress bar.

Listing 2-28. ch2_11b.html

```
<style>
    .ui-progressbar {
       height: 20px;
       width: 600px;
    }
    .ui-progressbar .ui-progressbar-value {
       background-image: url(images/pbar-ani.gif);
    }
</style>
```

The GIF image in Figure 2-20 gives a greater sense of the progress of the operation.

Figure 2-20. *A progress bar with an animated GIF gives a highly dynamic appearance to the web page*

■ **Note** Animated GIFs suitable for any kind of progress bar can be easily and safely obtained from the web site ajaxload (`www.ajaxload.info`). Simply choose the type of progress bar that you want to use, then the foreground and background colors, and the site automatically generates a preview of the animated GIF. If the image is to your liking, you can proceed with downloading it.

Otherwise, you can use the animated GIF (`pbar-ani.gif`) included in the code that accompanies this book, in the charts/images directory (you can find the code samples in the Source Code/Download area of the Apress web site [`www.apress.com/9781430262893`]).

Concluding Thoughts on the jQuery Library

By now, you are probably wondering why we started with a library (jQuery) that, apparently, does not have anything to do with the development of charts or with data visualization in general. You have seen that the jQuery UI library provides us with graphic elements, but its use is far from what you would expect when thinking about charts.

Actually, we had to start here. You have decided to work with the JavaScript language, with the aim of implementing graphic elements (which are nothing more than DOM elements) in web pages. At the heart of all this are the concepts introduced through this library. The selections, the chains of methods, the structure, the practice of using CSS styles—these are the basis of web programming and, even more so, of chart development. And, what better way to obtain these fundamentals than with the jQuery library?

As you progress through the book, you will find that most of the JavaScript libraries that you operate (jqPlot, Highcharts, and so on) must necessarily include the jQuery library. Even the D3 library, which does not use jQuery, has been structured in such a way as to be able to manage selections, chains of methods, structure—that is, the concepts that now form the basis of development in JavaScript.

That is why it is important to know jQuery.

Summary

Before starting to develop charts directly with JavaScript, it was necessary to introduce some fundamental tools that form the basis of the development of this type of code. In this second chapter, then, you were introduced to the **jQuery** and **jQuery UI** libraries. With jQuery, you learned how to manipulate DOM elements dynamically, through selections and chains of methods. With **jQuery UI**, you discovered how to enrich your pages with interactive graphic elements: the **jQuery UI widgets**.

In the next chapter, you will begin to implement your charts, using everything you have learned so far. You will start with the processing of incoming data, which you will do by parsing an HTML table.

CHAPTER 3

Simple HTML Tables

One of the simplest and most widely used forms of data display in an HTML page is the HTML table. Precisely because of its broad utility, the table was one of the first elements to be developed in HTML.

In this chapter, you will see how a table is structured and the HTML tags that implement it. The proper use of these tags can make the difference between a readable table and an incomprehensible one that will not allow you to perceive the underlying relationships.

Then, you will build a table containing data that you will use both here and in the next chapters. The purpose of this example is to understand the nature of a table and how data are structured within it. This is a crucial step toward being able to build the type of chart that best fits a particular data structure.

The charts responsible for the data visualization are fully implemented in JavaScript. Because the data that you need to represent in a chart are contained in an HTML table, you will see, in the second part of this chapter, how to implement a series of parsers in JavaScript language. Using the jQuery library, you will discover how easy it is to implement parsers that read specific data within the HTML table. These data, collected in arrays, are very accessible from the JavaScript language and easy to manipulate.

Creating a Table for Your Data

A table is simply a structure of nested tags, with the `<table>` tag as the root. The process for building this structure is not difficult, but it requires some forethought. First, you need to sketch out the table either on a piece of paper or at least, for those who are more familiar with tables, mentally. This helps determine the number of columns, rows, and headings that should be included in the table. Within the pair of tags `<table></table>`, you insert as many **rows** as are required as `<tr></tr>` pairs. Each `<tr>` tag creates a row inside the table. Then, you have to define the cells. Often, the top row contains headings, so you must specify them. You use the `<th></th>` pair to indicate the text that should be treated as a heading. For specifying ordinary cells, you use the `<td></td>` pair of tags. You must be careful to keep the number of cells consistent within the rows.

There are other tags that carry out functions designed to enrich the structure of a table. The `<caption>` tag is usually placed immediately after the opening `<table>` tag, and, when rendered, the content in the `<caption>` tag is shown above the table, in the middle. The tags `<thead>`, `<tfoot>`, and `<tbody>` improve the table structure significantly and provide additional hooks for Cascading Style Sheets (CSS) and JavaScript.

This is the basic procedure for building a table structure with HTML. Now, for a better understanding, you are going to create a table with a simple example.

Your Example's Goals

The Statistical Office of the Republic of Unhappy Children has recently published results regarding the number of balloons lost in space. You want to put this value in an HTML table.

Through this simple example, you will become familiar with the structuring of data within an HTML table, learning how to apply the jQuery selections in order to extract the data contained within.

Moreover, you will discover the role played by the CSS styles in terms of a table's graphics. By varying colors and text style, you can create a wide range of graphic themes. You will also see how to adjust the background colors by using gradients in order to give the cells in the table a three-dimensional appearance.

Listing 3-1 offers a set of data consisting of the number of balloons lost monthly for several countries over a period of six months.

Listing 3-1. ch3_01.html

```
<HTML>
<HEAD>
<TITLE>MyChart</TITLE>
</HEAD>
<BODY>
<table class="myTable">
<caption>Balloons Lost in Space</caption>
<thead>
    <tr>
    <td></td>
       <th>May 2013</th>
       <th>Jun 2013</th>
       <th>Jul 2013</th>
       <th>Aug 2013</th>
       <th>Sep 2013</th>
       <th>Oct 2013</th>
    </tr>
</thead>
<tbody>
    <tr>
    <th>USA</th>
       <td>12</td>
       <td>40</td>
       <td>75</td>
       <td>23</td>
       <td>42</td>
       <td>80</td>
    </tr>
    <tr>
    <th>Canada</th>
       <td>3</td>
       <td>22</td>
       <td>40</td>
       <td>27</td>
       <td>35</td>
       <td>21</td>
    </tr>
```

```
    <tr>
    <th>Australia</th>
       <td>60</td>
       <td>80</td>
       <td>16</td>
       <td>28</td>
       <td>33</td>
       <td>26</td>
    </tr>
    <tr>
    <th>Brazil</th>
       <td>46</td>
       <td>7</td>
       <td>14</td>
       <td>26</td>
       <td>36</td>
       <td>24</td>
    </tr>
</tbody>
<tfoot>
    <tr>
       <td colspan="7">Data from Statistical Office of the Republic of Unhappy Children</td>
    </tr>
</tfoot>
</table>
</BODY>
</HTML>
```

This simple HTML code gives the results shown in Figure 3-1.

Balloons lost in space

	May 2013	Jun 2013	Jul 2013	Aug 2013	Sep 2013	Oct 2013
USA	12	40	75	23	42	80
Canada	3	22	40	27	35	21
Australia	60	80	16	28	33	26
Brazil	46	7	14	26	36	24

Data from statistical office of the Republic of Unhappy Children

Figure 3-1. *A raw HTML table without any CSS style*

As you can see, this table is completely devoid of any graphics. It is presented as a series of strings in a table structure, but that is all. This is the moment CSS style comes into play.

Applying CSS to Your Table

CSS enriches the graphics mode of the table, making it more readable and, at the same time, more attractive. Each element of the HTML page can be referred to CSS style classes, and its graphical features can be adjusted by setting the attributes of these classes. In this way, you can use CSS to style the table as you like. It is possible to set a several style attributes for any element of an HTML page. This can be done thanks to CSS3.

■ **Note** This book does not discuss CSS styles in great detail, nor does it list all their possibilities. The subject is vast, and to treat it in depth could be misleading, for our purposes. However, the specific cases in this book list all the attributes that need to be set, thus allowing you to become more familiar with the vast world of CSS.

The definition of classes and their attributes is written inside the pair `<style></style>` in this way:

```
element.class {
    attribute: value;
}
```

Or, if you prefer, these definitions can be written in a CSS file that is subsequently included in one or more web pages. Thus, you can define the following CSS style classes for your table, writing the rows of code in the `<head>` section of the web page, as shown in Listing 3-2.

Listing 3-2. ch3_02a.html

```
<style type="text/css">
table.myTable caption {
    font-size: 14px;
    padding-bottom: 5px;
    font-weight: bold;
}
table.myTable {
    font-family: verdana, arial, sans-serif;
    font-size:11px;
    color:#333333;
    border-width: 1px;
    border-color: #666666;
    border-collapse: collapse;
}
table.myTable th {
    border-width: 1px;
    padding: 8px;
    border-style: solid;
    border-color: #666666;
    background-color: #bbdOda;
}
```

```
table.myTable td {
    border-width: 1px;
    padding: 8px;
    border-style: solid;
    border-color: #666666;
    background-color: #dedede;
}
</style>
```

Now, if you load the web page again, you can see the new layout of the table, as demonstrated in Figure 3-2.

Balloons lost in space

	May 2013	Jun 2013	Jul 2013	Aug 2013	Sep 2013	Oct 2013
USA	12	40	75	23	42	80
Canada	3	22	40	27	35	21
Australia	60	80	16	28	33	26
Brazil	46	7	14	26	36	24
Data from statistical office of the Republic of Unhappy Children						

Figure 3-2. *An HTML table in which you applied some CSS styles*

Looking at the table, you can see that the data are now much easier to read, and the appearance is also much more pleasant. The graphical representation above is only one of the infinite number of combinations possible. The attributes that can be set are so numerous that there are virtually no limits to the form your table can take. You can also integrate images and backgrounds to further increase the graphic capabilities.

Adding Color Gradation to Your Table

Now, you will continue to refine the appearance of your table. You have already made significant progress, but you can go further.

As you can see, the background color of the cells in the table is uniform, but you can create color gradients, assigning specific values to the CSS attributes. Because this can be somewhat difficult, the web page **Ultimate CSS Gradient Generator** (`http://www.colorzilla.com/gradient-editor`) can serve as a useful tool, helping you generate the gradients graphically by allowing you to select the colors and directions that they will take (see Figure 3-3).

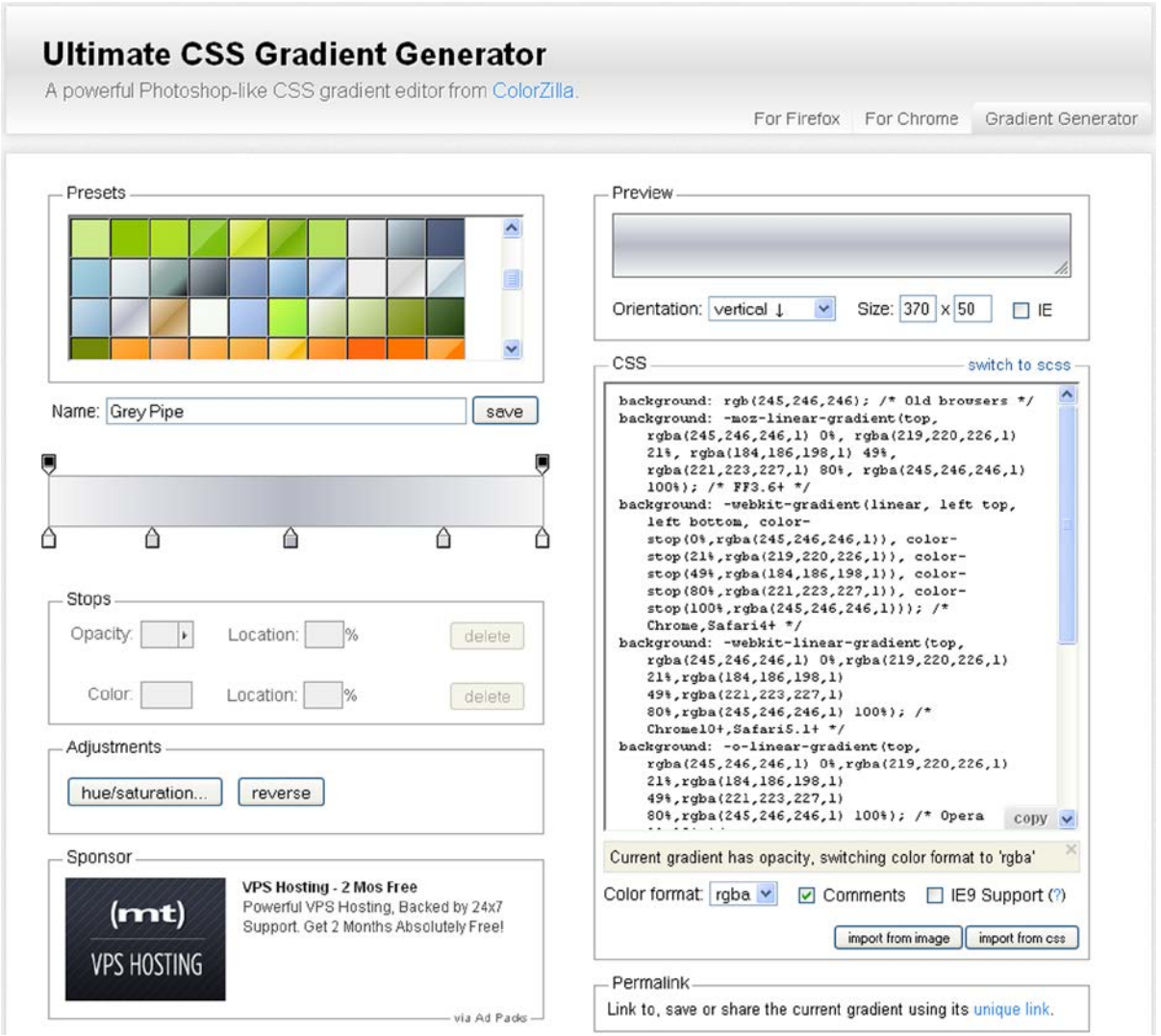


Figure 3-3. *Ultimate CSS Gradient Generator allows you to generate CSS gradients very easily*

From the Ultimate CSS Gradient web page, let us choose two presets to our liking, selecting the preset Grey 3D #4 for the gray cells and Blue Pipe #2 for the heading cells. Next, you copy the CSS attributes and paste them in your web page, as shown in Listing 3-3.

Listing 3-3. ch3_02b.html

```
table.myTable th {
    border-width: 1px;
    padding: 8px;
    border-style: solid;
    border-color: #666666;
    background: rgb(225,255,255); /* Old browsers */
```

```
background: -moz-linear-gradient(top, rgba(225,255,255,1) 0%,
   rgba(225,255,255,1) 7%,
   rgba(225,255,255,1) 12%,
   rgba(253,255,255,1) 12%,
   rgba(230,248,253,1) 30%,
   rgba(200,238,251,1) 54%,
   rgba(190,228,248,1) 75%,
   rgba(177,216,245,1) 100%); /* FF3.6+ */
background: -webkit-gradient(
   linear, left top, left bottom,
   color-stop(0%,rgba(225,255,255,1)),
   color-stop(7%,rgba(225,255,255,1)),
   color-stop(12%,rgba(225,255,255,1)),
   color-stop(12%,rgba(253,255,255,1)),
   color-stop(30%,rgba(230,248,253,1)),
   color-stop(54%,rgba(200,238,251,1)),
   color-stop(75%,rgba(190,228,248,1)),
   color-stop(100%,rgba(177,216,245,1))); /* Chrome,Safari4+ */
background: -webkit-linear-gradient(
   top,
   rgba(225,255,255,1) 0%,
   rgba(225,255,255,1) 7%,
   rgba(225,255,255,1) 12%,
   rgba(253,255,255,1) 12%,
   rgba(230,248,253,1) 30%,
   rgba(200,238,251,1) 54%,
   rgba(190,228,248,1) 75%,
   rgba(177,216,245,1) 100%); /* Chrome10+,Safari5.1+ */
background: -o-linear-gradient(
   top,
   rgba(225,255,255,1) 0%,
   rgba(225,255,255,1) 7%,
   rgba(225,255,255,1) 12%,
   rgba(253,255,255,1) 12%,
   rgba(230,248,253,1) 30%,
   rgba(200,238,251,1) 54%,
   rgba(190,228,248,1) 75%,
   rgba(177,216,245,1) 100%); /* Opera 11.10+ */
background: -ms-linear-gradient(
   top,
   rgba(225,255,255,1) 0%,
   rgba(225,255,255,1) 7%,
   rgba(225,255,255,1) 12%,
   rgba(253,255,255,1) 12%,
   rgba(230,248,253,1) 30%,
   rgba(200,238,251,1) 54%,
   rgba(190,228,248,1) 75%,
   rgba(177,216,245,1) 100%); /* IE10+ */
```

```
    background: linear-gradient(
       to bottom,
       rgba(225,255,255,1) 0%,
       rgba(225,255,255,1) 7%,
       rgba(225,255,255,1) 12%,
       rgba(253,255,255,1) 12%,
       rgba(230,248,253,1) 30%,
       rgba(200,238,251,1) 54%,
       rgba(190,228,248,1) 75%,
       rgba(177,216,245,1) 100%); /* W3C */
    filter: progid:DXImageTransform.Microsoft.gradient(
       startColorstr='#e1ffff', endColorstr='#b1d8f5',GradientType=0 ); /* IE6-9 */}
table.myTable td {
    border-width: 1px;
    padding: 8px;
    border-style: solid;
    border-color: #666666;
    background: rgb(242,245,246); /* Old browsers */
    background: -moz-linear-gradient(
       top,
       rgba(242,245,246,1) 0%,
       rgba(227,234,237,1) 37%,
       rgba(200,215,220,1) 100%); /* FF3.6+ */
    background: -webkit-gradient(
       linear, left top, left bottom,
       color-stop(0%,rgba(242,245,246,1)),
       color-stop(37%,rgba(227,234,237,1)),
       color-stop(100%,rgba(200,215,220,1))); /* Chrome,Safari4+ */
    background: -webkit-linear-gradient(
       top,
       rgba(242,245,246,1) 0%,
       rgba(227,234,237,1) 37%,
       rgba(200,215,220,1) 100%); /* Chrome10+,Safari5.1+ */
    background: -o-linear-gradient(
       top, rgba(242,245,246,1) 0%,
       rgba(227,234,237,1) 37%,
       rgba(200,215,220,1) 100%); /* Opera 11.10+ */
    background: -ms-linear-gradient(
       top,
       rgba(242,245,246,1) 0%,
       rgba(227,234,237,1) 37%,
       rgba(200,215,220,1) 100%); /* IE10+ */
    background: linear-gradient(
       to bottom,
       rgba(242,245,246,1) 0%,
       rgba(227,234,237,1) 37%,
       rgba(200,215,220,1) 100%); /* W3C */
    filter: progid:DXImageTransform.Microsoft.gradient(
       startColorstr='#f2f5f6', endColorstr='#c8d7dc',GradientType=0 ); /* IE6-9 */
```

You will immediately notice that the added code is quite extensive. Every browser has different specifications regarding the application of gradients on the attribute background. Because the user of your web site can request your page from any type of browser, you must cover all possibilities.

In Figure 3-4, you can see how the application of the gradients to the various cells gives the table a three-dimensional appearance.

Balloons lost in space

	May 2013	Jun 2013	Jul 2013	Aug 2013	Sep 2013	Oct 2013
USA	12	40	75	23	42	80
Canada	3	22	40	27	35	21
Australia	60	80	16	28	33	26
Brazil	46	7	14	26	36	24
Data from statistical office of the Republic of Unhappy Children						

Figure 3-4. *The CSS background attribute can be set with gradients to give the table a better appearance*

Adding Color Gradation to Your Table, Using Files

Color gradation is another style attribute that you can select for your table. The following example uses background images that allow the cells of the table to have color gradation. To achieve this, you must include two `.jpg` files in which you have drawn two color gradations: blue for heading cells and gray for cells with general values, as shown in Listing 3-4.

Listing 3-4. ch3_02c.html

```
table.myTable th {
    border-width: 1px;
    padding: 8px;
    border-style: solid;
    border-color: #666666;
    background:#b5cfd2 url('images/cell-blue.jpg');
}
table.myTable td {
    border-width: 1px;
    padding: 8px;
    border-style: solid;
    border-color: #666666;
    background:#dcddc0 url('images/cell-grey.jpg');
}
```

■ **Note** The two `.jpg` background files used to color the individual cells of the table can be found in the code that accompanies this book, in the Source Code/Download area of the Apress web site (`www.apress.com`).

Figure 3-5 illustrates the same HTML table, but this time, using the two images as backgrounds, the faded colors simulate shadows, giving the table a three-dimensional appearance.

Balloons lost in space

	May 2013	Jun 2013	Jul 2013	Aug 2013	Sep 2013	Oct 2013
USA	12	40	75	23	42	80
Canada	3	22	40	27	35	21
Australia	60	80	16	28	33	26
Brazil	46	7	14	26	36	24
Data from statistical office of the Republic of Unhappy Children						

Figure 3-5. *The same HTML table, but with another CSS style*

You have seen how to build an HTML table and how to assign it CSS styles. Now, let us look at the role that JavaScript will play throughout the book: data analysis and display in a chart. The input data, in this case, are represented in a table, but could be data obtained from a database or read from a file.

Parsing the Table Data

The previous chapter described the basics of the jQuery library. With the help of the functions provided by this library, in this chapter and the ones that follow, you will develop different kinds of charts displaying data in a table. But, where should you put the JavaScript code? Inside the `<script></script>` tag pairs, you will add the **$(document).ready()** function, writing your JavaScript code inside it in order to add all the events, or whatever else you want to include, before the window loads. Everything you write inside the brackets is executed as soon as the document object model (DOM) is registered by the browser. This lets you hide or show the elements of the page before it is open.

Importing the jQuery Library

To work with the jQuery functions, you have to import the jQuery library. It is not necessary that you download it from the jQuery web site and save it to your server; the web page can directly access the library from the distribution site. Listing 3-5 represents the starting point of the JavaScript code that you are going to write.

Listing 3-5. ch3_03a.html

```
<script src="http://code.jquery.com/jquery-1.9.1.min.js"></script></script>
<script>
$(document).ready(function(){
    //add your code here
});
</script>
```

■ **Note** If you are using the source code that accompanies the book, replace the reference to the jQuery library with

```
<script type="text/javascript" src="../src/jquery.min.js"></script>
```

Once the preliminaries are completed, you start immediately with the code. First, you will create a JavaScript object called `tableData` to hold all the information:

```
<script>
$(document).ready(function(){
    var tableData = {};
});
```

This variable will be used as a container for data parsed from the table. Brackets are used so as to consider it an object. You can add any property to this object by stating the `tableData.myNewProperty` and setting that property to whichever value you choose. In the same way, you can store the table elements in another variable:

```
$(document).ready(function(){
    var tableData = {};
    var table = $('table');
});
```

With this statement, you point to all the elements contained within the `<table></table>` tag pairs. With just a few words, you have made a selection.

xLabels

The first property you create in the `tableData` object is `xLabels`. It will contain the values for the labels on the x axis. These labels correspond to the cell content of the table headings. The labels are read in the order in which the data were collected, that is, from left to right. You find these values in the `<th>` tags (see Figure 3-6).

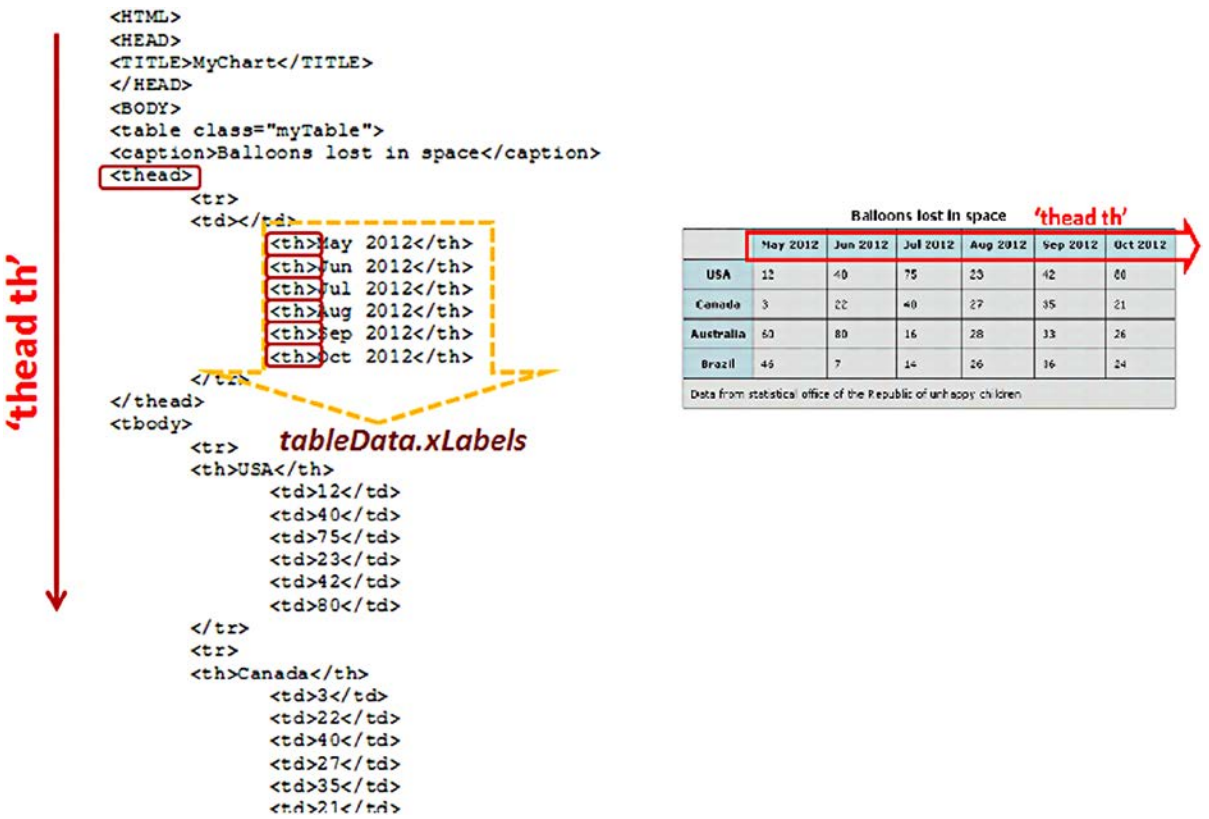


Figure 3-6. *Parsing the headings with the* `'thead th'` *as selector*

In Listing 3-6, `xLabels` is defined as an array, and, using the `each()` method, each `<th>` element is looped through, pushing its content into this array. Taking a closer look at the table, you can see that the `<th>` elements are nested within the `<thead>` tag. Therefore, a correct selection necessitates that you specify the hierarchy with `'thead th'`.

Listing 3-6. ch3_03b.html

```
$(document).ready(function(){
    var tableData = {};
    var table = $('table')
    tableData.xLabels = [];
    table.find('thead th').each(function(){
       tableData.xLabels.push( $(this).html() );
    });
});
```

Just for debugging, to see the contents of these arrays, you can use the console (see the section "Firebug and DevTools" in Chapter 1), calling the `log()` function to show the content of the variable passed as argument:

```
console.log(tableData.xLabels);
```

You have thus defined a new array, `xLabels`, containing the values shown in Figure 3-7.

Figure 3-7. *The content of the* `xLabels` *array, displayed in Firebug*

Extracting the Labels

You now need to extract the labels, referring the series of data from the table, as illustrated in Figure 3-8. In your table, you can identify these labels with the names of countries, shown on the left side.

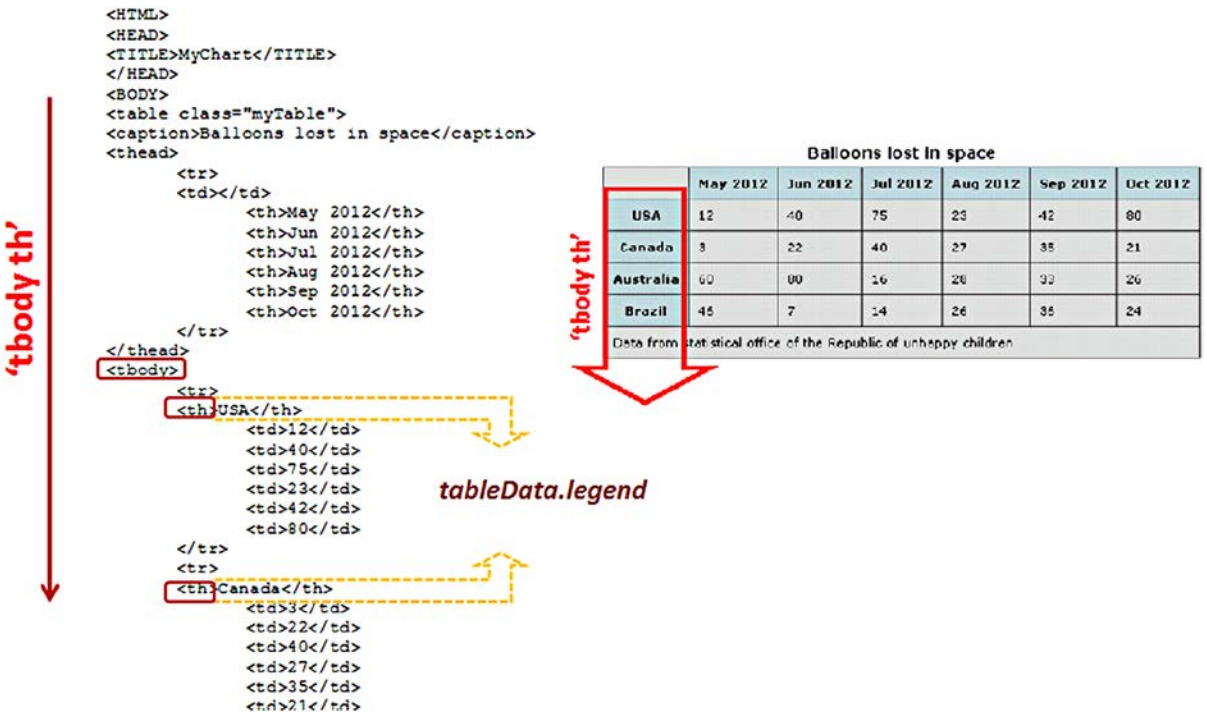


Figure 3-8. *Parsing the names of countries with* `'tbody th'` *as selector*

To capture the labels, you consider that they are grouped as `<th>` elements in the `<tbody>` group. Just as with `xLabels`, you define a new property of `tableData`: `legend` (see Listing 3-7). You assign this name because, in any kind of chart, the identifiers of the series are generally reported in a legend. So, you write a code similar to the previous one, this time using the selection `'tbody th'`.

Listing 3-7. ch3_03c.html

```
$(document).ready(function(){
    ...
    table.find('thead th').each(function(){
       tableData.xLabels.push( $(this).html() );
    });
```

```
    tableData.legend = [];
    table.find('tbody th').each(function(){
       tableData.legend.push( $(this).html() );
    });
});
```

Using the console on Firebug, you can see the content of the `legend` array, as displayed in Figure 3-9.

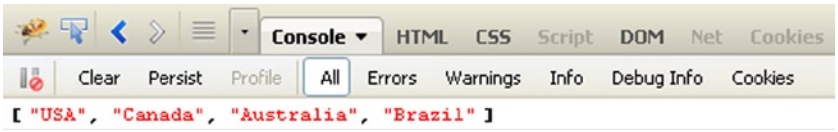


Figure 3-9. *The content of the* `legend` *array, displayed in Firebug*

By analyzing the points listed in the table, you can easily distinguish what will go on the x axis and what will go on the y axis. You collected data for several series (countries) at different times (months). It is easy to see that time will be represented on the x axis, using `xLabels` to fill its ticks; you assign a month for each tick. Furthermore, a whole series of values must be distributed on the y axis. You do not know the numeric values of the ticks on the y axis a priori, or how many ticks are necessary, but you need to calculate them. First, it is common practice, in these cases, to find the highest and lowest values in the data. You can select all the data with the `'tbody td'` selector, as shown in Listing 3-8.

Listing 3-8. ch3_03d.html

```
$(document).ready(function(){
    ...
    table.find('tbody th').each(function(){
       tableData.legend.push( $(this).html() );
    });
    var tmp = [];
    table.find('tbody td').each(function(){
       var thisVal = parseFloat( $(this).text() );
       tmp.push(thisVal);
    });
    if(Math.min.apply(null, tmp) > 0)
       tableData.minVal = 0;
    else
       tableData.minVal = Math.min.apply(null, tmp);
    tableData.maxVal = Math.max.apply(null, tmp);
});
```

You want the minimum value on the y axis to be equal to 0 and to assume lower values only if there are negative values in the table. In this case, there are only positive numbers, so the `minVal` is 0, and `maxVal` is 80.

Then, you use these two values to calculate the ticks on the y label. Based on these values, you should extend the y axis in a range from 0 to 80. Actually, it is best to extend the maximum value, so as not to have the maximum point of your data touching the top of the chart. You can multiply your maximum value by a coefficient (e.g., 10 percent). Let us correct the last line of Listing 3-8, introducing the coefficient, as demonstrated in Listing 3-9.

You have multiplied the maximum value, returned from the `Math.max.apply()` function, for the factor 1.1, thereby increasing the value by 10 percent (`max + 0.1*max = 1.1*max`).

Listing 3-9. ch3_03e.html

```
if(Math.min.apply(null, tmp) > 0)
    tableData.minVal = 0;
else
    tableData.minVal = Math.min.apply(null, tmp);
tableData.maxVal = 1.1 * Math.max.apply(null, tmp);
```

With regard to the number of ticks and their content, Listing 3-10 defines a `yLabels` array as a property of `tableData`. To quantify the number of ticks on the y axis so that it represents an optimal compromise, you must first determine what might be a suitable distance (in pixels) between one tick and the next. You divide the extent of the y axis by a number that represents the pixel distance between ticks. You may think that a distance of 30 pixels could be enough. The result is not an integer, so you need to round up.

Listing 3-10. ch3_03f.html

```
$(document).ready(function(){
    ...
    tableData.maxVal = 1.1 * Math.max.apply(null, tmp);
    tableData.yLabels = [];
    var yDeltaPixels = 30;
    var h = 360;
    var w = 460;
    var nTicks = Math.round(h / yDeltaPixels);
    var yRange = tableData.maxVal - tableData.minVal;
    var yDelta = Math.ceil(yRange / nTicks);
    var yVal = tableData.minVal;
    while( yVal < (tableData.maxVal - yDelta)){
       tableData.yLabels.push(yVal);
       yVal += yDelta;
    }
    tableData.yLabels.push(yVal);
    tableData.yLabels.push(tableData.maxVal);
});
```

If you investigate the contents of the `yLabels` array, you will find 12 values of y corresponding to 12 ticks, as shown in Figure 3-10. These labels will be displayed next to each tick.

Figure 3-10. *The content of the `yLabels` array, displayed in Firebug*

The values in the `yLabels` array are dependent on many factors, for instance, the dimension in which you want to represent the y axis (here, you chose 360 pixels), with each tick spaced 30 pixels apart: what you get here is 12 ticks. Also, if you want to know how many units of y correspond to the distance between one tick and the next, you calculate `yDelta` (rounded up), which, in this case, is eight. In fact, the values of `yLabels` are all multiples of eight.

dataGroups

The next property you need to create is `dataGroups`, a two-level array that contains all the values, grouped by series. Each series is an array of values, and `dataGroups` is an array of series. To group the data of different series, you use the `<tr>` tag. Each row in the table is, in fact, a series, and you can take all the values inside it because they are delimited by `<td>` tags, the cell tag. You capture the values, reading the data from left to right (see Figure 3-11), in conjunction with the direction of the times reported in the headings.

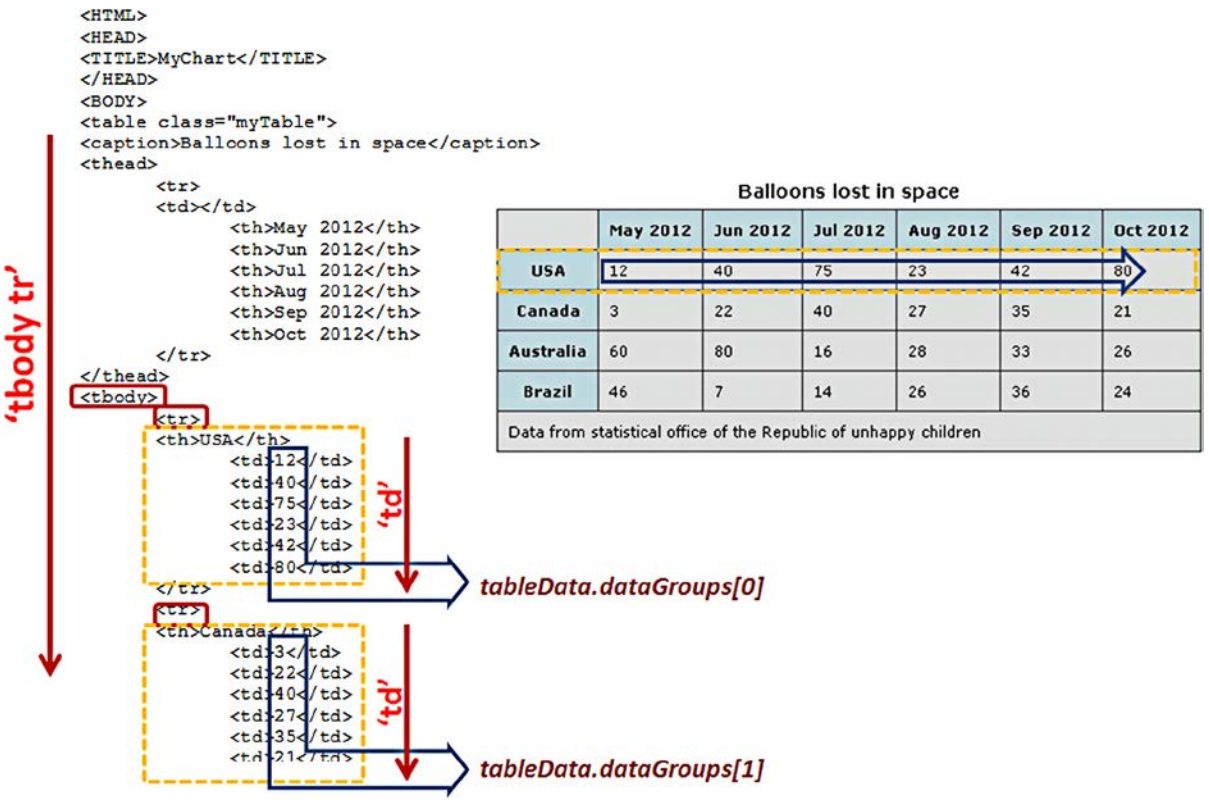


Figure 3-11. *Parsing of groups of data for multiseries*

Next, you loop through the rows of the table, using `'tbody tr'` as a selector (see Listing 3-11); for every step of this iteration, you must loop for each cell in order to reach the values.

Listing 3-11. ch3_03g.html

```
$(document).ready(function(){
    ...
    tableData.yLabels.push(tableData.maxVal);
    tableData.dataGroups = [];
```

```
    table.find('tbody tr').each(function(i){
       tableData.dataGroups[i] = [];
       $(this).find('td').each(function(){
          var tdVal = parseFloat( $(this).text() );
          tableData.dataGroups[i].push( tdVal );
       });
    });
});
```

In the end, you can see the four arrays (one for each series) contained in the `dataGroups` array, as shown in Figure 3-12.

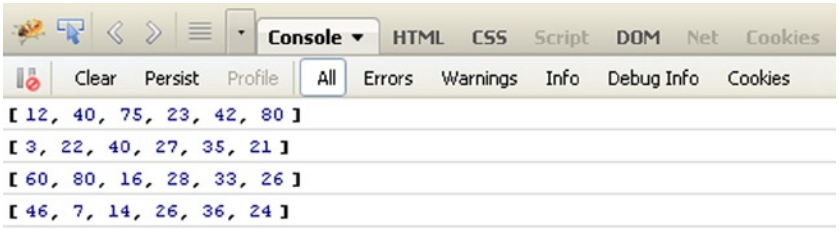


Figure 3-12. *The content of the `dataGroups` array, displayed in Firebug*

If you want to access the values contained in a particular series, you may do so as follows:

```
console.log(tableData.dataGroups[0]);
console.log(tableData.dataGroups[1]);
...
```

Ready for Implementing Graphics

Now that you have extracted all the data in the table and placed them in separate arrays, you are ready to begin to implement the graphics and convert these data into graphic elements.

This will be the topic of the next three chapters, in which you will see these data first represented in a line chart (Chapter 4), then in a bar chart (Chapter 5), and, finally, in a pie chart(Chapter 6).

Summary

The purpose of this chapter was to provide an introduction to the approach you must follow when you have a data structure to manipulate. With this chapter, you began to see how to develop your own library, based on the tools that the jQuery library provides.

You started with the creation of an **HTML table**, the most primitive form of data representation. Despite its simplicity, this table can be problematic if you do not set it well. Data representation in the form of a table has been chosen as a testing ground to begin looking at how jQuery is able to make selections of specific HTML elements on the page and, more specifically here, of tags that make up the HTML table. On this basis, you built a set of parsers to extract data from a complex structure (an HTML table, in this case, but, as you will see, from other types of structure, too). The data were thus separated into different groups, in a format that was easier to manipulate.

In the next chapter, you will begin to use the first graphic elements that the canvas offers, while continuing to use the jQuery library. As a first step, you will learn how to develop a **line chart**, using data obtained with the parsers developed in this chapter.

CHAPTER 4

Drawing a Line Chart

In the previous chapter, you built an HTML table as a sample of structured data and developed parsers in JavaScript to extract the data into arrays. In this chapter, you will build a line chart with JavaScript as one of the possible visualizations for the data in the table from Chapter 3.

Before you start to convert the data into graphic elements to achieve your line chart, you need a place in which to use it. Therefore, I will begin this chapter by introducing the canvas. You will see what this is and how to implement it, and, in the end, you will integrate it into your web page.

Once you understand what the canvas is, you can begin to actualize the elements that will make up your line chart. The first components that you are going to deal with are the axes, on which you will apply, according to the data to be represented, the ticks and the labels, followed by a grid as . background. Finally, you will complete your line chart by drawing the lines that represent the data read from the HTML table.

The last part of the chapter explains how to add other components less fundamental but very important in a line chart: the title and the legend.

Defining the Canvas

HTML5 technology allows you to define a drawing area called **canvas** on your web site. This area is defined as a real tag element, which occupies a defined area in a determined position, according to where the tag `<canvas>` is inserted.

In the previous chapter, you first developed an HTML table containing some data and then a set of parsers so that these data were accessible through arrays. Continuing from the point where you left off, you now add the `<canvas>` element, on which you can then draw all the elements of the line chart.

Thus, let us insert the `<canvas>` element within the `<body>` section, right where you want to place an area reserved for your chart (see Listing 4-1). Then, you have to insert the following tag before all the other tags related to the table, so that your chart is represented above the table when the page is loaded.

Listing 4-1. ch4_01.html

```
</HEAD>
<BODY>
<canvas id="myCanvas" width="500" height="400"> </canvas>
<table class="myTable">
    <caption>Balloons lost in space</caption>
    <thead>
...
```

In these tag statements, you have specified the size of the drawing area, too. To avoid having the drawing overlap the table, you can move the HTML table below the canvas. Therefore, in Cascading Style Sheets (CSS) styles, you need to define the new position of the table, as shown in Listing 4-2. The `top` attribute defines the distance from the top edge of the web page, whereas the `left` attribute defines the distance of the table from the left edge.

Listing 4-2. ch4_01.html

```
<style>
...
table.myTable {
    font-family: verdana,arial,sans-serif;
    font-size:11px;
    color:#333333;
    border-width: 1px;
    border-color: #666666;
    border-collapse: collapse;
    position: fixed;
    top: 450px;
    left: 20px;
}
...
</style>
```

Now that you have added the canvas to your page, you use it in JavaScript code. With jQuery, you have already seen that you can make a selection on any document object model (DOM) element, and the `<canvas>` tag is no exception. Then, applying a jQuery selection to this tag and assigning it to a variable (`canvas`, in this example), you can access the canvas using the JavaScript code (see Listing 4-3).

Because multiple canvases can be present in a web page, it will be necessary to distinguish them from each other. To this end, you assign the name '`myCanvas`' to the `id` attribute of the `<canvas>` element.

Listing 4-3. ch4_01.html

```
$(document).ready(function(){
    var canvas = $("#myCanvas");
    var tableData = {};
    var table = $('table');
    ...
});
```

To draw your charts, you will use the canvas two-dimensional drawing application programming interface (**canvas 2D drawing API)**, and so you have to set the canvas context on it. This context will provide you with all the objects and methods you need to draw and manipulate two-dimensional graphics in in the `canvas` drawing area. You set the context of your work on the canvas 2D API and assign it to the variable `ctx`, as demonstrated in Listing 4-4.

Listing 4-4. ch4_01.html

```
$(document).ready(function(){
    var canvas = $("#myCanvas");
    var ctx = canvas.get(0).getContext("2d");
    var tableData = {};
    var table = $('table');
    ...
});
```

■ **Note** If you were to try to use the canvas in an Internet Explorer browser below version 9, you would fail. Fortunately, Google has developed a library, ExplorerCanvas (`exCanvas`), that translates canvas commands into vector language markup (VML), a proprietary language supported by Internet Explorer. All you have to do is include the `excanvas.js` script in your web page. You can reference the script, using a conditional comment to make sure that only Internet Explorer sees and downloads this script.

```
<!--[if IE]><script src="excanvas.js"></script><![endif]-->
```

For further information, see the documentationpprovided on Google Code's ExplorerCanvas web page (http://excanvas.sourceforge.net).

Setting the Canvas

Let us take a look at the example you used in Chapter 3 to structure data in a table. You have a series for countries and some series of numeric values, with a time sequence for values taken month by month. Generally, it is preferable to use the x axis as the time axis; here, the y axis reports the number of balloons lost in space. You have seen that your data consist of four different series, one for each country, so there are four lines to represent in your chart. How do you distinguish between them? Usually, these are differentiated by color or by different graphic traits. For instance, a line may be drawn in a continuous manner, dashed, or dotted. In this case, you will use four different colors.

Thus, you need to set up an array of hexadecimal color values, which you will call `colors`, as shown in Listing 4-5.

Listing 4-5. ch4_01.html

```
$(document).ready(function(){
    ...
    var ctx = canvas.get(0).getContext("2d");
    var colors = ['#be1e2d', '#666699', '#92d5ea', '#ee8310'];
    var tableData = {};
    var table = $('table');
    ...
});
```

By default the canvas 2D API uses coordinates, starting from the top-left corner as the zero point, and the y coordinate grows with positive values, moving downward (see Figure 4-1). In contrast, the x coordinate is the same for both systems. Therefore, you must make some changes, applying the `translate()` method of the canvas 2D API to the context in order to change the position of the origin point of the axes, as presented in Listing 4-6.

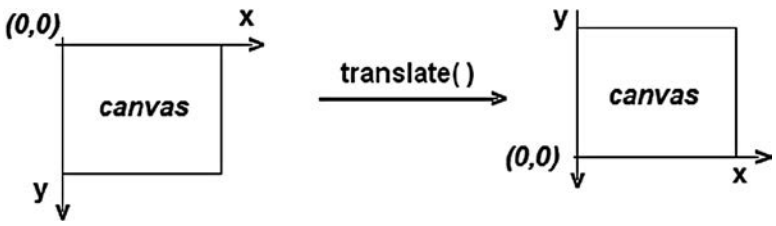


Figure 4-1. *Canvas translation to obtain the origin point in the bottom-left corner*

Listing 4-6. ch4_01.html

```
$(document).ready(function(){
    ...
    var colors = ['#be1e2d', '#666699', '#92d5ea', '#ee8310'];
    ctx.translate( 0, canvas.height() );
    var tableData = {};
    var table = $('table');
    ...
});
```

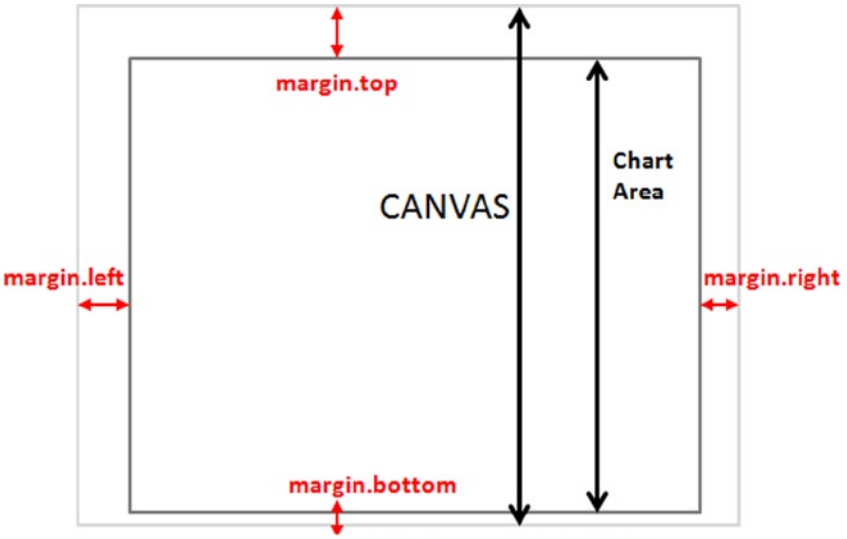
Now, the origin point of the chart is in the bottom-left corner. Unfortunately, it is not possible to reverse the direction of the y axis. Hence, you will write negative values of y to mean positive values, and vice versa. Normally, the chart would not take the whole drawing area, only the central area, bounded by margins. To add this border to your canvas, you draw a rectangle, which restricts the drawing area, excluding margins (see Listing 4-7). The `w` and `h` variables are the width and height, respectively, of the rectangle. These values are calculated taking the size of the canvas and margins into account.

The rectangle can be drawn using the `strokeRect()` function of the context, in which the first and the second argument are the (x, y) coordinates in the top-left corner, and the other two are the width and height of the rectangle.

Listing 4-7. ch4_01.html

```
$(document).ready(function(){
    ...
    ctx.translate( 0, canvas.height() );
    var margin = {top: 30, right: 10, bottom: 10, left: 30},
       w = canvas.width() - margin.left - margin.right,
       h = canvas.height() - margin.top - margin.bottom;
    ctx.strokeRect(margin.left, -margin.bottom, w, -h);
    var tableData = {};
    var table = $('table');
    ...
});
```

Figure 4-2 shows the areas covered by the canvas and the HTML table in the web page.



Balloons Lost in Space

	May 2013	Jun 2013	Jul 2013	Aug 2013	Sep 2013	Oct 2013
USA	12	40	75	23	42	80
Canada	3	22	40	27	35	21
Australia	60	80	16	28	33	26
Brazil	46	7	14	26	36	24

Data from Statistical Office of the Republic of Unhappy Children

Figure 4-2. *Representation of the parts of the canvas*

Drawing a Line Chart

Now that the drawing area is set up, you can begin to draw on it. Next, you will implement the line chart, component by component.

A line chart is a graph in which there are two Cartesian axes, x and y, which are assigned to two variables. On this two-dimensional graph, you will represent all (x, y) data points and then connect them with a line (see Figure 4-3).

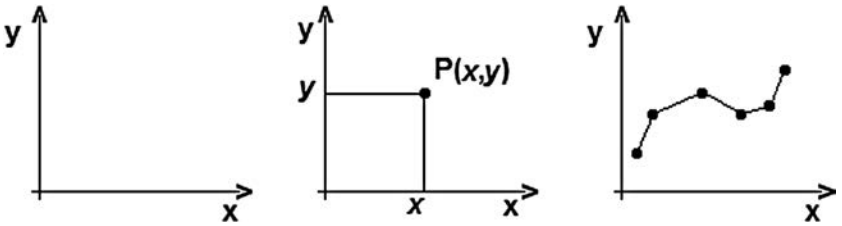


Figure 4-3. *(1) A cartesian chart, (2) a data point, (3) a line chart*

Therefore, before you begin to draw the lines on the xy plane, you will need to implement the two axes. Moreover, because you will be dealing with numeric values, it is also important to find a way to present these values in the right positions. Thus, you have to add scales to both axes, on which you will draw ticks and the corresponding values (tick labels). These scales will vary, depending on the ranges covered by the data on both axes.

Drawing Axes, Tick Labels, and the Grid

You start with the tick labels. In the previous chapter, you stored these values inside the `xLabels` and `yLabels` arrays. Now, place them along the axes of the chart.

Using native methods to write text directly on the canvas might seem to be the best choice, but it is actually preferable to use another approach. This is because the ability to manipulate and draw text on canvas is not yet well supported. The other method is to create dynamic text in HTML and then, using the power of CSS, modify the text to suit your needs. At first this may seem quite complex, but it will prove useful and practical.

You will create dynamic text with JavaScript embedded in unordered lists of `<ul>` tags containing all the tick labels, for both the x axis and the y axis. Thanks to CSS, you subsequently modify the labels' style and position in the HTML page, overlapping the canvas. You have to write the JavaScript code to generate HTML rows in the HTML page at the time of loading.

For example, to generate the ticks and labels on the x axis, use the code in Listing 4.8.

Listing 4-8. ch4_01.html

```
$(document).ready(function(){
    ...
    tableData.dataGroups = [];
    table.find('tbody tr').each(function(i){
       tableData.dataGroups[i] = [];
       $(this).find('td').each(function(){
          var tdVal = parseFloat( $(this).text() );
          tableData.dataGroups[i].push( tdVal );
       });
    });

    var xDelta = w / (tableData.xLabels.length -1 );
    var xlabelsUL = $('<ul class="labels-x"></ul>')
       .width(w)
       .height(h)
       .insertBefore(canvas);
    $.each(tableData.xLabels, function(i){
       var thisLi = $('<li><span class="label">' + this + '</span></li>')
```

```
            .prepend('<span class="line" />')
            .css('left', xDelta * i)
            .width(0)
            .appendTo(xlabelsUL);
         var label = thisLi.find('span.label');
         label.addClass('label');
    });
});
```

For debugging, you can use Firebug to see not only the content of a variable, but also the dynamically generated code. In fact, by selecting the HTML tab from the Firebug menu, you can see the entire tree structure of HTML code, including the part that you just generated dynamically for displaying the tick labels on the x axis (see Figure 4-4).

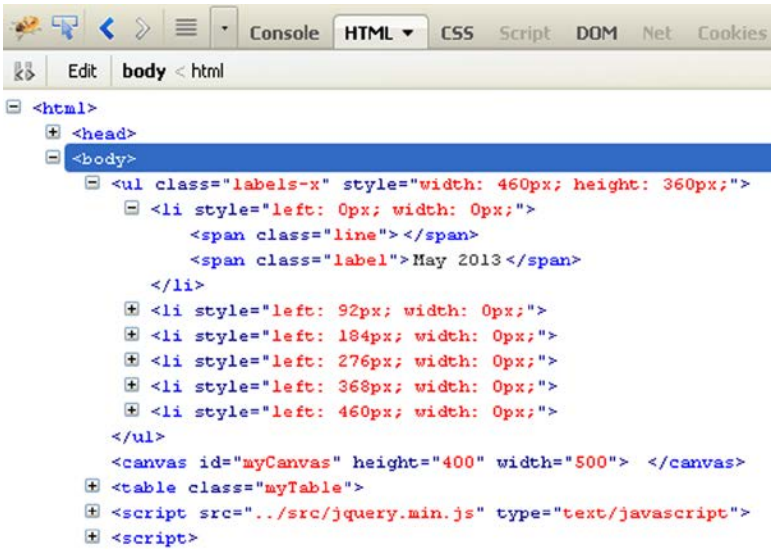


Figure 4-4. *By selecting the HTML tab in Firebug, you can see the HTML structure dynamically generated*

As you will note in Figure 4-4, each item on the list contains two spans: one for the tick and one for the tick label. Now, you need to do the same thing with the y labels, by writing the code in Listing 4.9.

Listing 4-9. ch4_01.html

```
$(document).ready(function(){
    ...
    $.each(tableData.xLabels, function(i){
       var thisLi = $('<li><span class="label">'+this+'</span></li>')
          .prepend('<span class="line" />')
          .css('left', xDelta * i)
          .width(0)
          .appendTo(xlabelsUL);
       var label = thisLi.find('span.label');
       label.addClass('label');
    });
```

```
    var yScale = h / yRange;
    var liBottom = h / (tableData.yLabels.length-1);
    var ylabelsUL = $('<ul class="labels-y"></ul>')
       .width(w)
       .height(h)
       .insertBefore(canvas);
    $.each(tableData.yLabels, function(i){
        var thisLi = $('<li><span>' + this + '</span></li>')
          .prepend('<span class="line"  />')
          .css('bottom', liBottom * i)
          .prependTo(ylabelsUL);
       var label = thisLi.find('span:not(.line)');
       var topOffset = label.height() / -2;
       if(i == 0){ topOffset = -label.height(); }
       else if(I == tableData.yLabels.length - 1){ topOffset = 0; }
       label
          .css('margin-top', topOffset)
          .addClass('label');
    });
});
```

But, if you load the page, you will see only two unordered lists, reporting all the values contained in the `xLabels` and `yLabels` arrays (see Figure 4-5).

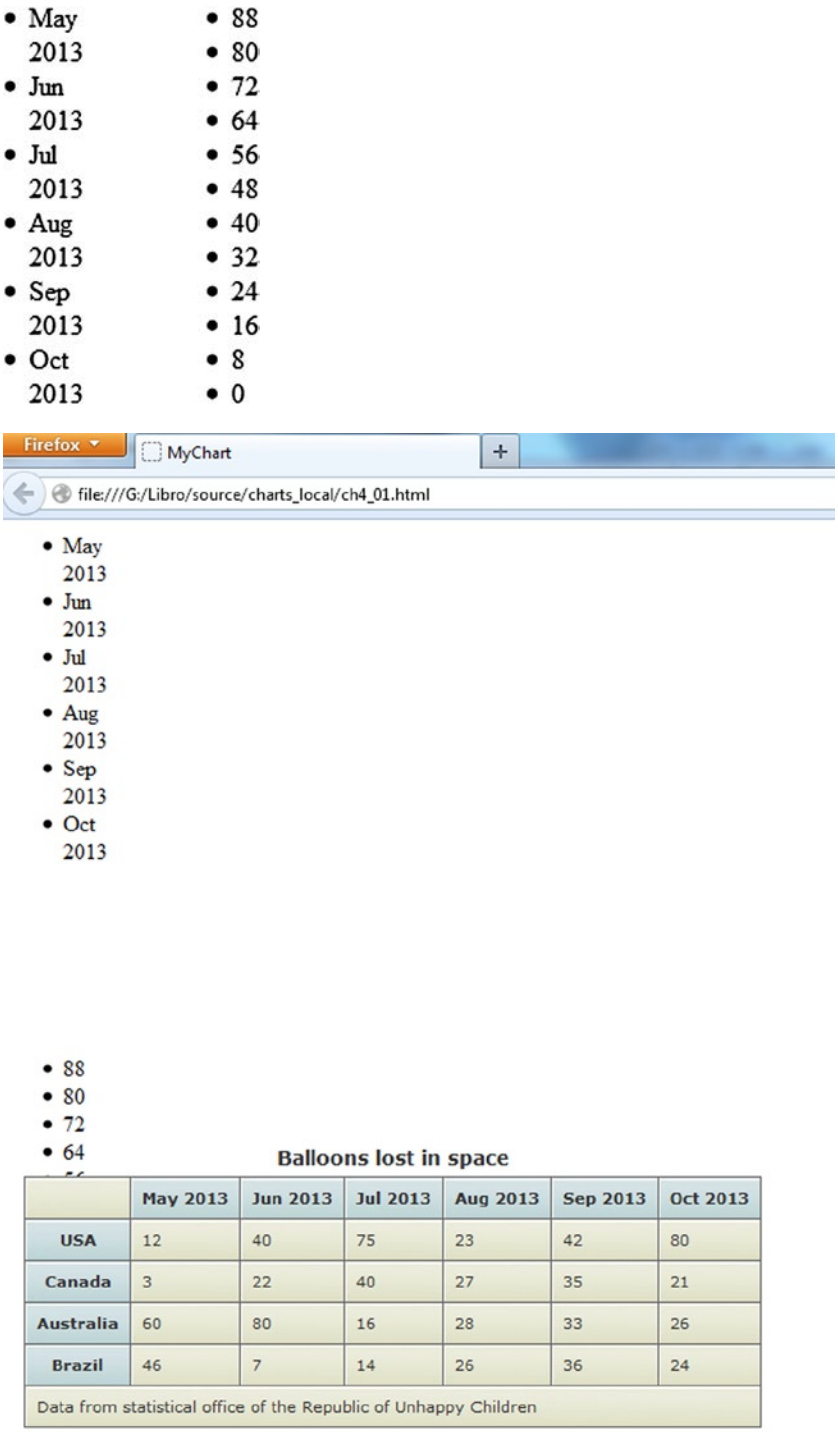


Figure 4-5. *If you use just JavaScript, the browser displays only two unordered lists partially covered by the table (the rectangle containing the canvas is not visible.*

This is miles from what you could have been imagining. In fact, these do not look like ticks and labels at all. It is here that you can see the power of CSS and how it can totally change the appearance of a list, adapting it to your purposes. Looking at the two unordered lists, the first thing you will want to do is delete the black dot alongside each item. So, let us write the CSS style class definition, referring to the <ul> and <li> tags as shown in Listing 4-10.

Listing 4-10. ch4_01.html

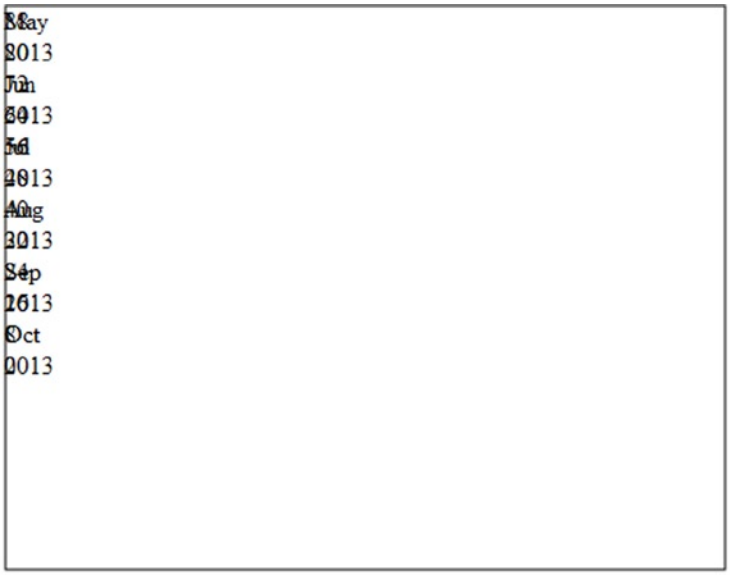
```
<style>
    ...
table.myTable td {
    border-width: 1px;
    padding: 8px;
    border-style: solid;
    border-color: #666666;
    background:#dcddc0 url('images/cell-grey.jpg');
}
ul, .li {
    margin: 0;
    padding: 0;
}
</style>
```

Then, you add an additional CSS style class that will let you write the lists not at the top of the page, but above the canvas (see Listing 4-11). To attach styles to the HTML elements you generated, you referred to them with a specific class name inserted in the <ul> tag: <*ul class = "labels-x"*> and <*ul class = "labels-y "*>. Consequently, in the CSS, you will refer to them by these names, preceded by a '.'.

Listing 4-11. ch4_01.html

```
<style>
    ...
ul, .li {
    margin: 0;
    padding: 0;
}
.labels-x, .labels-y {
    position: absolute;
    left: 37;
    top: 37;
    list-style: none;
}
</style>
```

If you load the page now, you will see the changes illustrated in Figure 4-6. The two unordered lists have become two columns of text labels, covering the left side of the canvas. In addition, the rectangle representing the drawing area of the canvas is shown at the top of the web page (the space occupied by the lists is now free).



Balloons Lost in Space

	May 2013	Jun 2013	Jul 2013	Aug 2013	Sep 2013	Oct 2013
USA	12	40	75	23	42	80
Canada	3	22	40	27	35	21
Australia	60	80	16	28	33	26
Brazil	46	7	14	26	36	24

Data from Statistical Office of the Republic of Unhappy Children

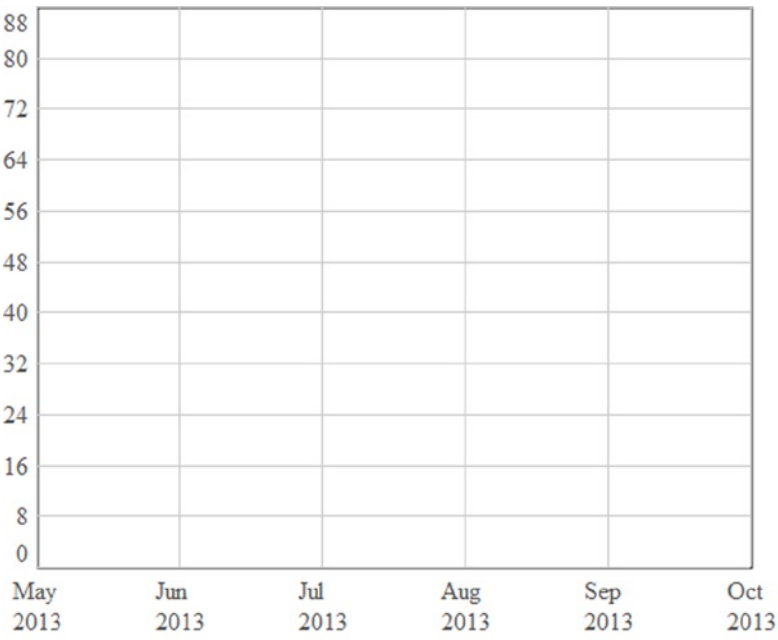
Figure 4-6. *The unordered lists have become two columns of text labels laid over the canvas*

But, by adding more CSS classes, as in Listing 4-12, you can complete your work, finally creating an xy plane with ticks, ticks labels, and a grid background, as shown in Figure 4-7.

Listing 4-12. ch4_01.html

```
<style>
    ...
.labels-x, .labels-y {
    position: absolute;
    left: 37;
    top: 37;
    list-style: none;
}
```

```
.labels-x li {
    position: absolute;
    bottom: 0;
    height: 100%;
}
.labels-x li span.label {
    position: absolute;
    color: #555;
    top: 100%;
    margin-top: 5px;
    left:-15;
}
.labels-x li span.line{
    position: absolute;
    border: 0 solid #ccc;
    border-left-width: 1px;
    height: 100%;
}
.labels-y li {
    position: absolute;
    bottom: 0;
    width: 100%;
}
.labels-y li span.label {
    position: absolute;
    color: #555;
    right: 100%;
    margin-right: 5px;
    width: 100px;
    text-align: right;  }
.labels-y li span.line {
    position: absolute;
    border: 0 solid #ccc;
    border-top-width: 1px;
    width: 100%;
}
</style>
```



Balloons Lost in Space

	May 2013	Jun 2013	Jul 2013	Aug 2013	Sep 2013	Oct 2013
USA	12	40	75	23	42	80
Canada	3	22	40	27	35	21
Australia	60	80	16	28	33	26
Brazil	46	7	14	26	36	24

Data from Statistical Office of the Republic of Unhappy Children

Figure 4-7. *The grid lines for a line chart*

That is incredible! Starting with two simple unordered lists, it is possible to obtain a chart with a grid, x and y labels, and ticks—all with only a few CSS style statements.

Drawing Lines on the Chart

In the previous chapter, you stored all the data in a property of the `tableData` object called `dataGroups`. This array has many series, so you need to iterate through and for each of them to plot a line on the grid just drawn. You have already figured the increment to effect on x for every data point for the `xLabels` array, and this value is in the `xDelta` variable, so you do not need to calculate it again.

You can, therefore, determine the thickness of the lines, setting the `lineWidth` property of the context to 5 pixels, as shown in Listing 4-13.

Listing 4-13. ch4_02.html

```
$(document).ready(function(){
    ...
    $.each(tableData.xLabels, function(i){
       ...
    });
    ctx.lineWidth = 5;
});
```

Now, you are ready to loop through the `dataGroups` array and draw the lines. For each series, you need to start from the first data point, so you use the function `ctx.moveTo(0,-point[i])` to move the context to the first y value (the x is 0 because it is the first point). The variable `i` is the index to iterate between the series. With the function `ctx.strokeStyle(),` you assign each line a color, which is previously defined in an array. Then, you effectively begin to draw the line, creating a path in which each step is defined by a line starting from the current point and pointing to a new one with (x, y) coordinates. This is expressed by the `ctx.lineTo(x,y)` function. The new x coordinate is the current x value plus the `xDelta` calculated before. The new y coordinate is the next value in the array identified by `points[j]`, where `j` is the index of the value contained in each series. When the loop has completed the j values, the path will end with the `closePath()` function. The path will then go to the next series,effecting increments of the variable `i`, and the loop will be repeated until all series are completed. All this can be better expressed in Listing 4-14.

Listing 4-14. ch4_02.html

```
$(document).ready(function(){
    ...
    ctx.lineWidth = 5;
    for(var i in tableData.dataGroups){
       var points = tableData.dataGroups[i];
       ctx.moveTo(0, -points[i]);
       ctx.strokeStyle = colors[i];
       ctx.beginPath();
       var xVal = margin.left;
       for(var j in points){
          var relY = (points[j] * h /tableData.maxVal) + 10;
          ctx.lineTo(xVal, -relY);
          xVal += xDelta;
       }
       ctx.stroke();
       ctx.closePath();
    }
});
```

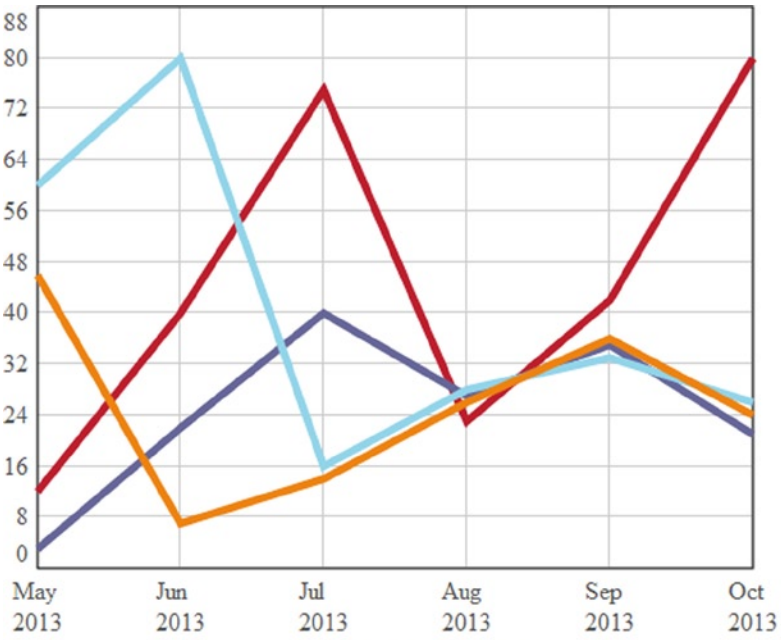
Moreover, in order to avoid having the lines in your chart appear under the grid, instead of above it, you add the CSS statement for the canvas, as presented in Listing 4-15.

Listing 4-15. ch4_02.html

```
<style>
    ...
canvas {
    position: relative;
}
```

```
ul, .li {
    margin: 0;
    padding: 0;
}
    ...
</style>
```

Figure 4-8 shows your line chart, achieved with the combination of JavaScript code and CSS styles.



Balloons Lost in Space

	May 2013	Jun 2013	Jul 2013	Aug 2013	Sep 2013	Oct 2013
USA	12	40	75	23	42	80
Canada	3	22	40	27	35	21
Australia	60	80	16	28	33	26
Brazil	46	7	14	26	36	24

Data from Statistical Office of the Republic of Unhappy Children

Figure 4-8. *The line chart representing the data in the table*

Adding a Legend

Each series in the line chart is represented by a different color, but at the moment, you have no reference linking color and country, so you must draw a legend. A legend is a small table reporting the identifiers of a charts' series. You have already created a `legend` array containing the names of the countries. You loop through this array, generating an unordered list, as shown in Listing 4-16.

Listing 4-16. ch4_03.html

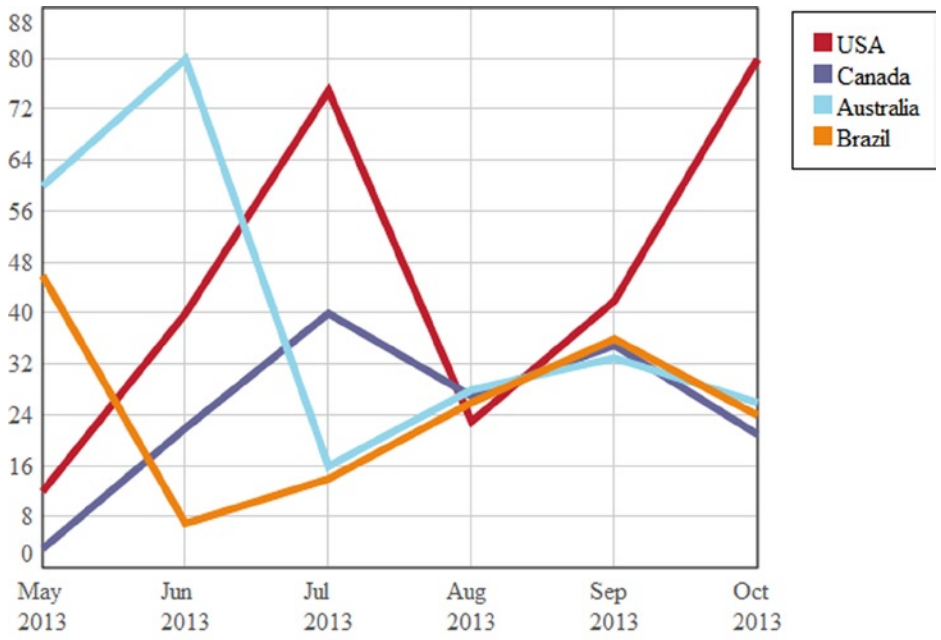
```
$(document).ready(function(){
    ...
    ctx.lineWidth = 5;
    for(var i in tableData.dataGroups){
       ...
    }
    var legendList = $('<ul class="legend"></ul>')
       .insertBefore(canvas);
    for(var i in tableData.legend){
       $('<li>' + tableData.legend[i] + '</li>')
       .prepend('<span style="background: ' + colors[i] + '" />')
       .appendTo(legendList);
    }
});
```

Because with the legend you have added another element to the canvas, you need to define as well the CSS styles for it, as demonstrated in Listing 4-17.

Listing 4-17. ch4_03.html

```
<style>
    ...
.labels-y li span.line {
    position: absolute;
    border: 0 solid #ccc;
    border-top-width: 1px;
    width: 100%;
}
.legend {
    list-style: none;
    position: absolute;
    left: 520px;
    top: 40px;
    border: 1px solid #000;
    padding: 10px;
}
.legend li span {
    width: 12px;
    height: 12px;
    float: left;
    margin: 3px;
}
</style>
```

A legend thus appears on the right side of the chart, as shown in Figure 4-9.



Balloons Lost in Space

	May 2013	Jun 2013	Jul 2013	Aug 2013	Sep 2013	Oct 2013
USA	12	40	75	23	42	80
Canada	3	22	40	27	35	21
Australia	60	80	16	28	33	26
Brazil	46	7	14	26	36	24

Data from Statistical Office of the Republic of Unhappy Children

Figure 4-9. *Adding a legend reporting the name of the countries*

Adding a Title

An often overlooked and omitted element is the title. This element, however, adds readability to the chart. Let us then add a text at the top of the page, using the content of the caption element of the table (see Listing 4-18).

Listing 4.18. ch4_04.html

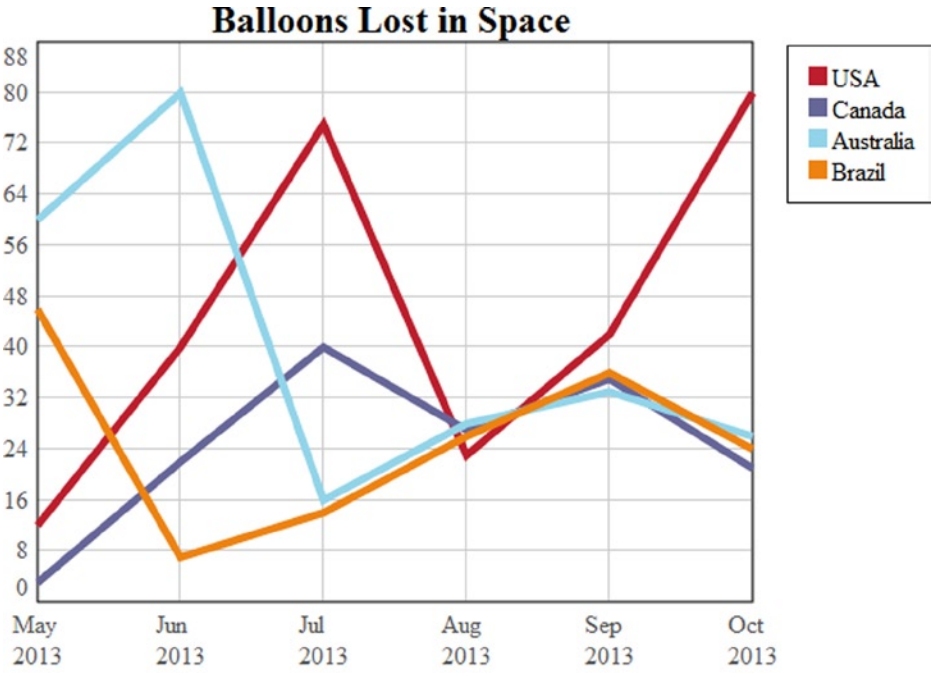
```
$(document).ready(function(){
    ...
    for(var i in tableData.legend){
       $('<li>'+ tableData.legend[i] +'</li>')
       .prepend('<span style="background: '+ colors[i] +'" />')
       .appendTo(legendList);
    }
    $('<div class="chart-title">' + table.find('caption').html() + '</div>')
       .insertBefore(canvas);});
```

At the same time, you add CSS style attributes to the new class `chart-title`, as shown in Listing 4-19. Defining the `top` and `left` attributes (similar to what you did with the table), you can set the position of the title in the web page.

Listing 4.19. ch4_04.html

```
<style>
    ...
.legend li span {
    width: 12px;
    height: 12px;
    float: left;
    margin: 3px;
}
.chart-title {
    font-size: 24;
    font-weight: bold;
    position: absolute;
    left: 150px;
    top: 10px;
    width: 100%;
}
</style>
```

Figure 4-10 illustrates your completed line chart.



Balloons Lost in Space

	May 2013	Jun 2013	Jul 2013	Aug 2013	Sep 2013	Oct 2013
USA	12	40	75	23	42	80
Canada	3	22	40	27	35	21
Australia	60	80	16	28	33	26
Brazil	46	7	14	26	36	24

Data from Statistical Office of the Republic of Unhappy Children

Figure 4-10. *Adding a title above the line chart*

Hiding the Table

You started showing data, using an HTML table, and then parsed all the values inside the table, using JavaScript code, and realized the values by building a line chart. At this point, showing the table may seem superfluous, so you can hide it from the user who will load the page. A good approach is to remove it from the page entirely, using jQuery (see Listing 4-20). After all the data are parsed, you select the table, and its contents, and hide it by applying the `hide()` function to the selection. It is very simple!

Listing 4.20. ch4_04.html

```
$(document).ready(function(){
    ...
    $('<div class="chart-title">' + table.find('caption').html() + '</div>')
       .insertBefore(canvas);
    $('table').hide();
});
```

Summary

In this chapter, you began to look at how to make a **line chart** graphically by using the jQuery library and manipulating the data that you extracted with the parser developed in the previous chapter. You also saw that it is possible to give a shape to the grid and axis label with CSS styles.

In the next chapter, you will learn how to use the same data to generate another type of chart that is very common: the **bar chart**.

CHAPTER 5

Drawing a Bar Chart

In the previous chapter, you created a line chart, using the data in an HTML table. However, the same data can be represented by other charts, including the well-known bar chart. With this type of data visualization, you still have the two axes (x and y), but x, instead of being represented as a continuous scale of values, is used to represent groups, which may or may not follow an increasing trend.

In this chapter, you will continue to use the code written in the previous chapter, and you will see how, by making a few changes to the code (very few!), you can convert your line chart into a bar chart. Once you have implemented the data parsers and set all the graphic elements, such as the axes, the tick labels, and the grid, you will find that it is very easy to switch from a line chart to a bar chart.

Drawing a Bar Chart

The first thing to change is the way in which you calculate the `xDelta` variable. In this case, the x axis has lost its meaning and serves only to group data in different sectors. The HTML table has six different dates, and so you divide the x axis into six segments. In the line chart these six dates were in correspondence with the grid lines, giving you five segments. Let us modify the `xDelta` variable, taking these differences into account, as shown in Listing 5-1.

Listing 5-1. ch5_01a.html

```
$(document).ready(function(){
    ...
    table.find('tbody tr').each(function(i){
       tableData.dataGroups[i] = [];
       $(this).find('td').each(function(){
          var tdVal = parseFloat( $(this).text() );
          tableData.dataGroups[i].push( tdVal );
       });
    });
    var xDelta = w / (tableData.xLabels.length);
    var xlabelsUL = $('<ul class="labels-x"></ul>')
       .width(w)
       .height(h)
       .insertBefore(canvas);
...
});
```

Now, delete the rows that are no longer necessary, shown in bold in Listing 5-2.

Listing 5-2. ch5_01b.html

```
$(document).ready(function(){
    ...
//delete the following rows
    ctx.lineWidth = 5;
    for(var i in tableData.dataGroups){
       var points = tableData.dataGroups[i];
       ctx.moveTo(0,-points[i]);
       ctx.strokeStyle = colors[i];
       ctx.beginPath();
       var xVal = margin.left;
       for(var j in points){
          var relY = (points[j]*h/tableData.maxVal) + 10;
          ctx.lineTo(xVal,-relY);
          xVal += xDelta;
       }
       ctx.stroke();
       ctx.closePath();
    } //end delete
...
});
```

In their place, you can now write the code (see Listing 5-3).

Listing 5-3. ch5_01c.html

```
$(document).ready(function(){
    ...
    $.each(tableData.yLabels, function(i){
       var thisLi = $('<li><span>'+this+'</span></li>')
          .prepend('<span class="line"  />')
          .css('bottom',liBottom*i)
          .prependTo(ylabelsUL);
       var label = thisLi.find('span:not(.line)');
       var topOffset = label.height()/-2;
       if(i == 0){ topOffset = -label.height(); }
       else if(i== tableData.yLabels.length-1){ topOffset = 0; }
       label
          .css('margin-top', topOffset)
          .addClass('label');
    });

    var barGroupMargin = 4;
    for(var i in tableData.dataGroups){
       ctx.beginPath();
       var n = tableData.dataGroups.length;
       var lineWidth = (xDelta - barGroupMargin * 2 ) / n;
       var strokeWidth = lineWidth - (barGroupMargin * 2);
       ctx.lineWidth = strokeWidth;
```

```
        var points = tableData.dataGroups[i];
        var xVal = (xDelta - n * strokeWidth - (n - 1) * (lineWidth - strokeWidth)) / 2;
        for(var j in points){
            var relX = margin.left + (xVal - barGroupMargin) + (i * lineWidth) + lineWidth / 2;
            ctx.moveTo(relX, -margin.bottom);
            var relY = margin.bottom + points[j] * h / tableData.maxVal;
            ctx.lineTo(relX, -relY);
            xVal += xDelta;
        }
        ctx.strokeStyle = colors[i];
        ctx.stroke();
        ctx.closePath();
    }

    var legendList = $('<ul class="legend"></ul>')
        .insertBefore(canvas);
    for(var i in tableData.legend){
        $('<li>'+ tableData.legend[i] +'</li>')
        .prepend('<span style="background: '+ colors[i] +'" />')
        .appendTo(legendList);
    }
...
});
```

Because you are working with a bar chart rather than a line chart, the x axis reports the categories. This means that the x labels should no longer appear in correspondence with ticks, but at the center of each interval, delimited by two ticks. To do this quickly and easily, you have to edit the `margin-left` Cascading Style Sheets (CSS) attribute relative to the x labels, using the `css()` jQuery function directly (see Listing 5-4).

Listing 5-4. ch5_01d.html

```
$(document).ready(function(){
    ...
    $.each(tableData.xLabels, function(i){
        var thisLi = $('<li><span class="label">' + this + '</span></li>')
            .prepend('<span class="line" />')
            .css('left', xDelta * i)
            .width(0)
            .appendTo(xlabelsUL);
        var label = thisLi.find('span.label');
        label
            .css('margin-left', '40px')
            .addClass('label');
    });
    ...
});
```

The resulting bar chart is illustrated in Figure 5-1.

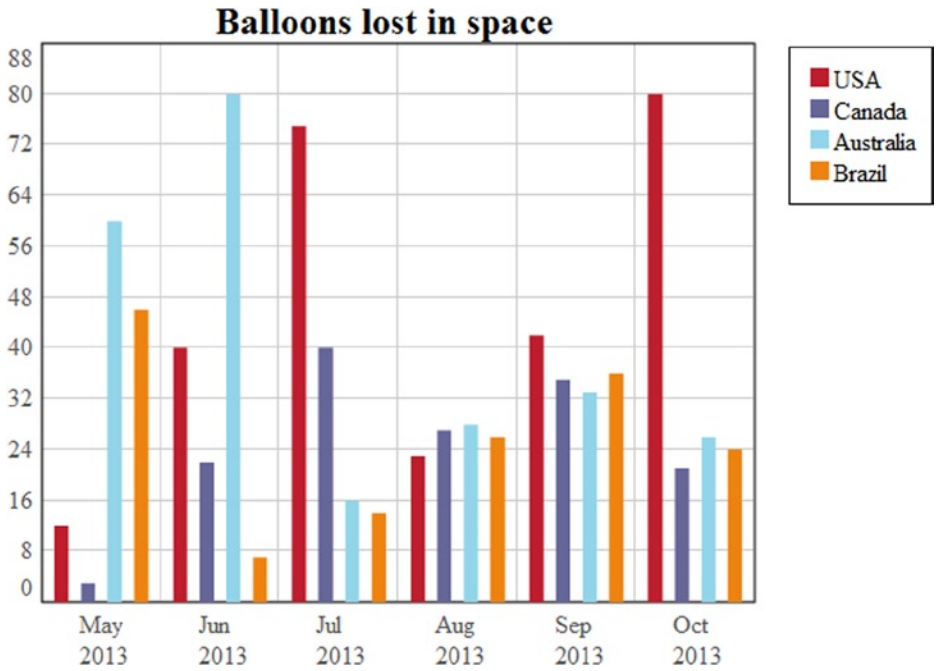


Figure 5-1. *A bar chart representing the data in the table*

Summary

In this chapter, you saw how easy it is to create a **bar chart**, using what you learned in the previous chapters. In the next chapter, you will implement another type of chart that uses neither a grid, nor axes, but cirular sectors: the **pie chart**. You will also discover how to insert animations in response to certain events, such as mouse clicks, to increase the interactivity of your chart.

CHAPTER 6

Drawing a Pie Chart

Similar to what you have done in the last two chapters, in this chapter, you will learn to build a pie chart, using the data contained in your HTML table.

Starting from the point at which you parsed all the data (see Chapter 3), you will discover how to implement this interesting kind of object and explore how to create your first animations.

Drawing a Pie Chart

This type of chart is quite different from the previous two (line and bar charts). It consists of a circle (a pie) that represents the sum of all the values (see Figure 6-1). This pie is divided into slices of different colors, one for each series present in the data. The size of the slices is proportional to the totality of all the values in the series, in relation to the sum of all the values in the pie. Thus, each slice expresses the percentage that each series contributes to the sum with its value.

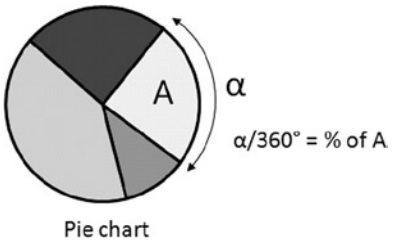


Figure 6-1. *A pie chart*

Let us see how to implement the particular features of this kind of chart.

Setting the Canvas

As previously stated, in developing your pie chart, you will start with the code that was obtained at the end of Chapter 3, that is, with the data contained in an HTML table and a number of parsers implemented to extract data from the table and place them within different arrays. Listing 6-1 shows the status of the code at the end of Chapter 3.

Listing 6-1. ch6_01.html

```
<HTML>
<HEAD>
<TITLE>MyChart</TITLE>
<style type="text/css">
table.myTable caption {
   font-size: 14px;
   padding-bottom: 5px;
   font-weight: bold;
}
table.myTable {
    font-family: verdana,arial,sans-serif;
    font-size:11px;
    color:#333333;
    border-width: 1px;
    border-color: #666666;
    border-collapse: collapse;
}
table.myTable th {
    border-width: 1px;
    padding: 8px;
    border-style: solid;
    border-color: #666666;
    background:#b5cfd2 url('images/cell-blue.jpg');
}
table.myTable td {
    border-width: 1px;
    padding: 8px;
    border-style: solid;
    border-color: #666666;
    background:#dcddc0 url('images/cell-grey.jpg');
}
</style>
</HEAD>
<BODY>
<canvas id="myCanvas" width="500" height="400"> </canvas>
<table class="myTable">
    <caption>Balloons Lost in Space</caption>
    <thead>
       <tr>
          <td></td>
           <th>May 2013</th>
           <th>Jun 2013</th>
           <th>Jul 2013</th>
           <th>Aug 2013</th>
           <th>Sep 2013</th>
           <th>Oct 2013</th>
       </tr>
    </thead>
```

```
    <tbody>
       <tr>
          <th>USA</th>
          <td>12</td>
          <td>40</td>
          <td>75</td>
          <td>23</td>
          <td>42</td>
          <td>80</td>
       </tr>
       <tr>
          <th>Canada</th>
          <td>3</td>
          <td>22</td>
          <td>40</td>
          <td>27</td>
          <td>35</td>
          <td>21</td>
       </tr>
       <tr>
          <th>Australia</th>
          <td>60</td>
          <td>80</td>
          <td>16</td>
          <td>28</td>
          <td>33</td>
          <td>26</td>
       </tr>
       <tr>
          <th>Brazil</th>
          <td>46</td>
          <td>7</td>
          <td>14</td>
          <td>26</td>
          <td>36</td>
          <td>24</td>
       </tr>
    </tbody>
    <tfoot>
       <tr><td colspan="7">Data from Statistical Office of the Republic of Unhappy
          Children</td></tr>
    </tfoot>
</table>
<script src="http://code.jquery.com/jquery-1.9.1.min.js"></script>
<script>
$(document).ready(function(){
    var tableData = {};
    var table = $('table');
    tableData.xLabels = [];
    table.find('thead th').each(function(){
       tableData.xLabels.push( $(this).html() );
    });
```

```
    tableData.legend = [];
    table.find('tbody th').each(function(){
       tableData.legend.push( $(this).html() );
    });
    var tmp = [];
    table.find('tbody td').each(function(){
       var thisVal = parseFloat( $(this).text() );
       tmp.push(thisVal);
    });
    if(Math.min.apply(null, tmp) > 0)
       tableData.minVal = 0;
    else
       tableData.minVal = Math.min.apply(null, tmp);
    tableData.maxVal = 1.1 * Math.max.apply(null, tmp);
    tableData.yLabels = [];
    var yDeltaPixels = 30;
    var nTicks = Math.round(h / yDeltaPixels);
    var yRange = tableData.maxVal - tableData.minVal;
    var yDelta = Math.ceil(yRange / nTicks);
    var yVal = tableData.minVal;
    while( yVal < (tableData.maxVal - yDelta)){
       tableData.yLabels.push(yVal);
       yVal += yDelta;
    }
    tableData.yLabels.push(yVal);
    tableData.yLabels.push(tableData.maxVal);
    tableData.dataGroups = [];
    table.find('tbody tr').each(function(i){
       tableData.dataGroups[i] = [];
       $(this).find('td').each(function(){
          var tdVal = parseFloat( $(this).text() );
          tableData.dataGroups[i].push( tdVal );
       });
    });
});
</script>
</BODY>
</HTML>
```

So, let us begin with the definition of the canvas context. Unlike with line and bar charts, you do not need to translate the context, because in pie charts there are no axes to represent; you define the margins and the size of the canvas instead (see Listing 6-2). Note the values passed into `strokeRect()`: the y values must be passed as positive (there is no translation).

Listing 6-2. ch6_01.html

```
$(document).ready(function(){
    var canvas = $("#myCanvas");
    var ctx = canvas.get(0).getContext("2d");
    var colors = ['#be1e2d', '#666699', '#92d5ea', '#ee8310'];
    var margin = {top: 30, right: 10, bottom: 10, left: 30},
       w = canvas.width() - margin.left - margin.right,
```

```
        h = canvas.height() - margin.top - margin.bottom;
    ctx.strokeRect(margin.left,margin.top,w,h);
    var tableData = {};
    var table = $('table');
    tableData.xLabels = [];
    ...
});
```

Implementing the Pie Chart

Next, you define the center point of the circle of the pie chart (see Listing 6-3). Assuming that it corresponds to the center of the drawing area, you define two variables, `center_x` and `center_y`, representing the coordinates of the center point, and from there you begin to build your chart. Once you have defined a `pieMargin` to establish a distance between the edges of the circle and margins, you define the radius. The size of the drawing area depends on the difference between the `pieMargin` (which is fixed) and the center of the circle (which is dynamic). Thus, the radius will change, adapting to the size of the drawing area. All these values must be calculated, taking the margin into account.

Listing 6-3. ch6_01.html

```
$(document).ready(function(){
    ...
    ctx.strokeRect(margin.left,margin.top,w,h);
    var pieMargin = margin.top + 30;
    var center_x = Math.round(w / 2) + margin.left;
    var center_y = Math.round(h / 2) + margin.top;
    var radius = center_y - pieMargin;
    var counter = 0.0;
    var tableData = {};
    var table = $('table');
    tableData.xLabels = [];
    ...
});
```

The pie represents the sum of all the values written in the table. You therefore define a function called `dataSum` and then a variable with the same name, as shown in Listing 6-4.

Listing 6-4. ch6_01.html

```
$(document).ready(function(){
    ...
    table.find('tbody tr').each(function(i){
       tableData.dataGroups[i] = [];
       $(this).find('td').each(function(){
          var tdVal = parseFloat( $(this).text() );
          tableData.dataGroups[i].push( tdVal );
       });
    });
    var dataSum = function(){
       var dataSum = 0;
       for(var i in tableData.dataGroups){
```

```
            var points = tableData.dataGroups[i];
            for(var j in points){
               dataSum += points[j];
            }
         }
         return dataSum;
      }
      var dataSum = dataSum();
});
```

You then write an unordered list <ul> tag, with every list item consisting of a label shown next to its corresponding slice, as demonstrated in Listing 6-5.

Listing 6-5. ch6_01.html

```
$(document).ready(function(){
    ...
    var dataSum = dataSum();
    var labels = $('<ul class="labels"></ul>')
       .insertBefore(canvas);
});
```

Now, as Listing 6-6 illustrates, you can draw the slices of the pie, one by one.

Listing 6-6. ch6_01.html

```
$(document).ready(function(){
    ...
    var labels = $('<ul class="labels"></ul>')
       .insertBefore(canvas);
    for(var i in tableData.dataGroups){
       var sum = 0;
       var points = tableData.dataGroups[i];
       for(var j in points){
          sum += points[j];
       }
       var fraction = sum/dataSum;
       ctx.beginPath();
       ctx.moveTo(centerx, centery);
       ctx.arc(centerx, centery, radius,
          counter * Math.PI * 2 - Math.PI * 0.5,
          (counter + fraction) * Math.PI * 2 - Math.PI * 0.5, false);
       ctx.lineTo(centerx, centery);
       ctx.closePath();
       ctx.fillStyle = colors[i];
       ctx.fill();
       var sliceMiddle = (counter + fraction/2);
       var distance = radius * 1.2;
       var labelx = Math.round(centerx + Math.sin(sliceMiddle * Math.PI * 2) *
          (distance));
       var labely = Math.round(centery - Math.cos(sliceMiddle * Math.PI * 2) *
          (distance));
       var leftPlus = (labelx < centerx) ? '40' : '0' ;
```

```
        var percentage = parseFloat((fraction*100).toFixed(2));
        var labelval = percentage + "%";
        var labeltext = $('<span class="label">' + labelval +'</span>')
           .css('font-size', radius / 8)
           .css('color', colors[i]);
        var label = $('<li class="label-pos"></li>')
           .appendTo(labels)
           .css({left: labelx-leftPlus, top: labely})
           .append(labeltext);
        counter+=fraction;
    }
});
```

Let us break down the listing:

- You are using a path to draw each slice, employing the `ctx.arc()` function to account for the circle's edge.
- The `counter` is the cumulative value directly correlated to the percentage covered by the slices already drawn, and it is a value from 0 to 1. When this value is 1, you have drawn 100 percent of the pie.
- The variable `fraction` is the percentage covered by each single slice, and the corresponding percentage is stored in the variable with the same name: `percentage`.
- Next, you define the `sliceMiddle` variable, which represents the angle (percentage) of the bisector of each slice. You use the `sliceMiddle` value to place the label at the right angle, which is perfectly in the middle of the slice.
- You choose to place the label reporting the percentage just outside the slices (but nothing would prevent you from representing them inside). The `distance` variable is how far from the center of the pie you decide to place the label. Here, you select a value corresponding to 120 percent of the radius.
- Each label is written in the web page as a dynamic `<li>` tag, in a font size that is proportional to the size of the pie and a color corresponding to the country (series).

If you want to see how these dynamically generated rows are structured, you can use Firebug or DevTools (see Chapter 1), selecting the HTML tab from the menu. Figure 6-2 shows the particular structure that you just generated dynamically.

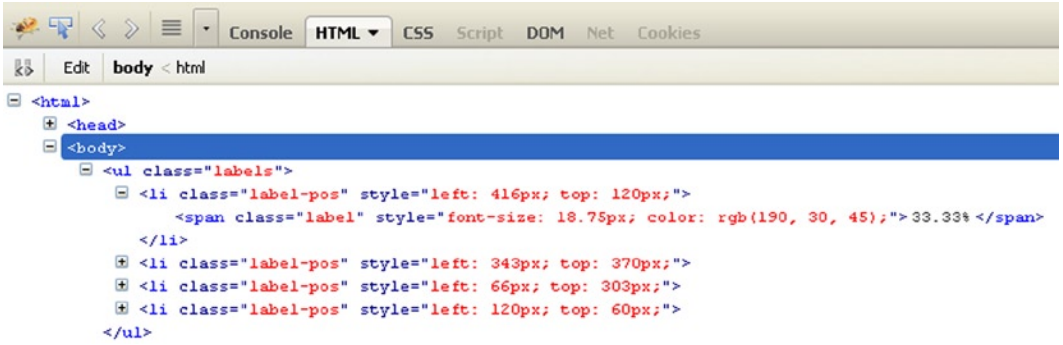


Figure 6-2. *Using Firebug, it is possible to see the HTML structure dynamically generated*

Completing the Pie Chart

To complete your pie chart, you need to add a title. You can take it directly, from inside the table, making a selection on `caption`, as presented in Listing 6-7.

Listing 6-7. ch6_01.html

```
$(document).ready(function(){
    ...
    for(var i in tableData.dataGroups){
       ...
    }
    $('<div class="chart-title">'+table.find('caption').html() + '</div>')
       .insertBefore(canvas);
});
```

The last but not least element to add to your chart is a legend. This is especially important for this type of chart, as there are no references to the data series that correspond to the colors of the slices. Thus, you must add this element, as shown in Listing 6-8.

Listing 6-8. ch6_01.html

```
$(document).ready(function(){
    ...
    $('<div class="chart-title">'+table.find('caption').html() + '</div>')
       .insertBefore(canvas);
    var legendList = $('<ul class="legend"></ul>')
       .insertBefore(canvas);
    for(var i in tableData.legend){
       $('<li>'+ tableData.legend[i] +'</li>')
          .prepend('<span style="background: '+ colors[i] +'" />')
          .appendTo(legendList);
    }
});
```

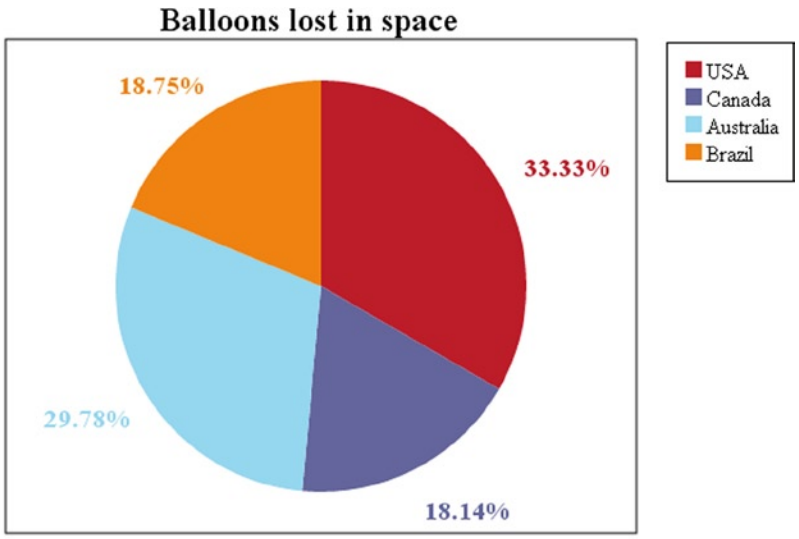
Also, you have added many classes to be referenced in the Cascading Style Sheets (CSS) styles statements (see Listing 6-9). Furthermore, you must place the canvas in top part of the page, so you need to position the table below it, setting the `top` and `left` CSS attributes of the `table.myTable` class.

Listing 6-9. ch6_01.html

```
<style>
    ...
table.myTable {
    font-family: verdana,arial,sans-serif;
    font-size:11px;
    color:#333333;
    border-width: 1px;
    border-color: #666666;
    border-collapse: collapse;
```

```
    position: fixed;
    top: 450px;
    left: 20px;
}
    ...
ul, .li {
    margin: 0;
    padding: 0;
}
.labels {
    list-style: none;
}
.label-pos, label {
    position: absolute;
    margin-left: 0px;
    margin-top: 0px;
    padding:0;
}
.label { display: block;
    color: #fff;
    font-weight: bold;
    font-size: 1em;
}
.chart-title {
    font-size: 24;
    font-weight: bold;
    position: absolute;
    left: 150px;
    top: 10px;
    width: 100%;
}
.legend {
    list-style: none;
    position: absolute;
    left:520;
    top: 40;
    border: 1px solid #000;
    padding: 10px;
}
.legend li span {
    width: 12px;
    height: 12px;
    float: left;
    margin: 3px;
}
</style>
```

Finally, you obtain your pie chart presented in Figure 6-3.



Balloons lost in space

	May 2013	Jun 2013	Jul 2013	Aug 2013	Sep 2013	Oct 2013
USA	12	40	75	23	42	80
Canada	3	22	40	27	35	21
Australia	60	80	16	28	33	26
Brazil	46	7	14	26	36	24

Data from statistical office of the Republic of Unhappy Children

Figure 6-3. *The pie chart representing the data in the table*

Adding Effects

Now that you have obtained your pie chart, you will investigate further, learning how to improve the appearance of your chart by adding interesting effects to it. You will see how you can manipulate the color of the slices by adding gradients. You will also discover how to animate the slices in order to create a pie chart that is interactive.

Adding a Gradient Effect

You drew your pie chart with slices filled with uniform color, but you can make some changes. For instance, you can add a gradient effect to the slices. To accomplish this, you need to replace

```
ctx.fillStyle = colors[i];
```

with the rows in Listing 6-10.

Listing 6-10. ch6_02.html

```
$(document).ready(function(){
    ...
    ctx.beginPath();
    ctx.moveTo(center_x, center_y);
    ctx.arc(center_x, center_y, radius,
       counter * Math.PI * 2 - Math.PI * 0.5,
       (counter + fraction) * Math.PI * 2 - Math.PI * 0.5, false);
    ctx.lineTo(center_x, center_y);
    ctx.closePath();
    var sliceGradientColor = "#ddd";
    var sliceGradient = ctx.createLinearGradient( 0, 0, w, h );
    sliceGradient.addColorStop( 0, sliceGradientColor );
    sliceGradient.addColorStop( 1, colors[i]);
    ctx.fillStyle = sliceGradient;
    ctx.fill();
    var sliceMiddle = (counter + fraction/2);    ...
});
```

For the purpose of these examples, the display of the HTML table is no longer required, so you can hide it, selecting it and then chaining the `hide()` function (see Listing 6-11).

Listing 6-11. ch6_02.html

```
$(document).ready(function(){
    ...
    $('table').hide();
});
```

As you can see in Figure 6-4, you have chosen a white color as gradient color overlapping the existing colors, but you could choose any color just by writing a red-green-blue (RGB) hexadecimal into the `sliceGradientColor`.

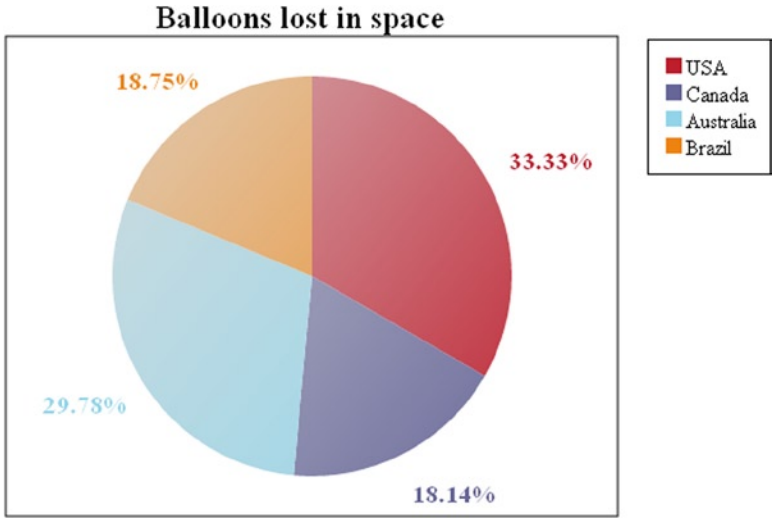


Figure 6-4. *Adding a white gradient to your pie chart*

Adding a Better Gradient Effect

But, to be honest, you are not very happy with your result, so you make additional changes to achieve a better gradient effect (see Listing 6-12). This time, you choose a dark color, almost black, as a color gradient, assigning a very dark gray to the `sliceGradientColor` variable. Furthermore, you separate the slices by creating spaces of division, so as to enhance the gradient effect even more. Instead of drawing the slices separated, a better choice is to draw the spaces in between, creating a white border. Thus, you define the `sliceBorderWidth` variable, with which you can adjust the division size, and the `sliceBorderStyle` variable, to set the white color.

Listing 6-12. ch6_03.html

```
$(document).ready(function(){
    ...
    ctx.beginPath();
    ctx.moveTo(center_x, center_y);
    ctx.arc(center_x, center_y, radius,
       counter * Math.PI * 2 - Math.PI * 0.5,
       (counter + fraction) * Math.PI * 2 - Math.PI * 0.5, false);
    ctx.lineTo(center_x, center_y);
    ctx.closePath();
    var sliceGradientColor = "#222";
    var sliceBorderStyle = "#fff";
    var sliceBorderWidth = 4;
    var sliceGradient = ctx.createLinearGradient(0, 0, w*.7, h*.7);
    sliceGradient.addColorStop(0, sliceGradientColor);
    sliceGradient.addColorStop(1, colors[i]);
    ctx.fillStyle = sliceGradient;
    ctx.fill();
    ctx.lineWidth = sliceBorderWidth;
    ctx.strokeStyle = sliceBorderStyle;
    ctx.stroke();
    var sliceMiddle = (counter + fraction / 2);
    ...
});
```

Figure 6-5 shows the same pie chart, but with a black gradient that gives more depth to the slices.

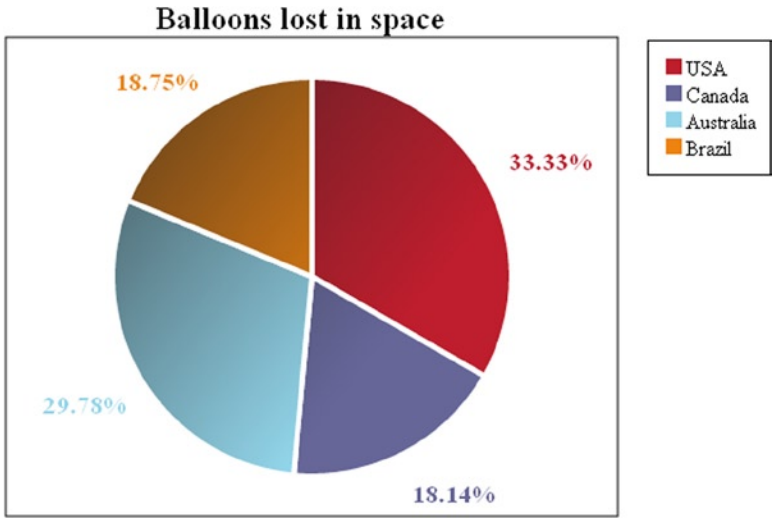


Figure 6-5. *Adding a black gradient and space divisions to your pie chart*

Creating a Pie Chart with a Slice Pulled Out

Pies are usually eaten, and, to entice you to do so, often the baker will pull a slice away from the rest of the pie to better display his or her wares. Joking aside, if, among the different series represented in a pie chart, you want to highlight one in particular, sometimes the relevant slice is pulled out.

Now, suppose you are interested in the third series; you can pull this slice out. To do this, first you will need to add two new variables in the loop through the slices: `startAngle` and `endAngle`, as shown in Listing 6-13.

Listing 6-13. ch6_04.html

```
$(document).ready(function(){
    ...
    for(var i in tableData.dataGroups){
       var sum = 0;
       var points = tableData.dataGroups[i];
       for(var j in points){
          sum += points[j];
       }
       var fraction = sum / dataSum;
       var startAngle = counter * Math.PI * 2 - Math.PI * 0.5;
       var endAngle = (counter + fraction) * Math.PI * 2 - Math.PI * 0.5;
       ctx.beginPath();
       ctx.moveTo(center_x, center_y);
       ...
    }
    ...
});
```

Inside the `for()` loop, you have to write code that is active only for the third slice—that is, when the index `i` is 2 (see Listing 6-14).

Listing 6-14. ch6_04.html

```
$(document).ready(function(){
    ...
    for(var i in tableData.dataGroups){
       var sum = 0;
       var points = tableData.dataGroups[i];
       for(var j in points){
          sum += points[j];
       }
       var fraction = sum / dataSum;
       var startAngle = counter * Math.PI * 2 - Math.PI * 0.5;
       var endAngle = (counter + fraction) * Math.PI * 2 - Math.PI * 0.5;
       if(i == 2){
          var currentPullOutDistance = 20;
          var maxPullOutDistance = 25;
          var ratio = currentPullOutDistance/maxPullOutDistance;
          var midAngle = (startAngle + endAngle) / 2;
          var actualPullOutDistance = currentPullOutDistance *
             (Math.pow( 1 - ratio, .8 ) + 1);
          var startx = centerx + Math.cos(midAngle) * actualPullOutDistance;
          var starty = centery + Math.sin(midAngle) * actualPullOutDistance;
          ctx.beginPath();
          ctx.moveTo(startx, starty);
          ctx.arc(startx, starty, radius, startAngle,endAngle, false);
          ctx.lineTo(startx, starty);
          ctx.closePath();
       }else{
          ctx.beginPath();
          ctx.moveTo(centerx, centery);
          ctx.arc(centerx, centery, radius, startAngle,endAngle, false);
          ctx.lineTo(centerx, centery);
          ctx.closePath();
       }
       var sliceGradientColor = "#222";
       var sliceBorderStyle = "#fff";
       var sliceBorderWidth = 4;
       ...
    }
    ...
});
```

Figure 6-6 shows the chart with a slice pulled out.

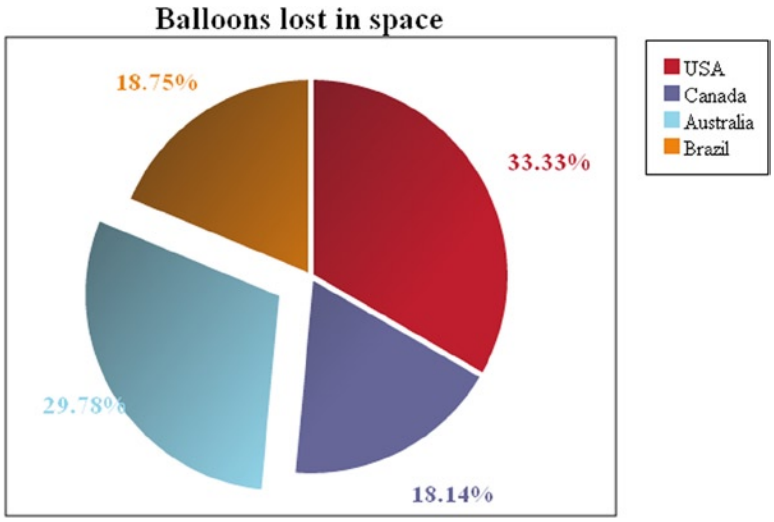


Figure 6-6. *The pie chart with a slice pulled out*

In this example, you have seen how to pull a slice out of the pie. This is the first step toward the goal that you have set for yourself, that is, to make your pie chart interactive. The next step will be to create an animation in which you can see a slice being extracted from the chart.

Inserting an Animation to Pull Out the Slice

With JavaScript there are no limits to the changes that you can make to your chart, except your imagination. An additional touch to your chart might be to create an animation. For example, when the page has just been loaded, you could display the pie chart still intact and then, gradually, pull out the slice you want to highlight with an animation.

The code that you are developing is gradually becoming more complex; therefore, it is necessary to begin grouping lines of code together, according to their functionality, thus avoiding a repetitive and less readable code. Let us start with the code for the gradient effect. First, you write the `sliceGradient()` function, which manages the color gradient effect. This function accepts only one argument: the color gradient overlapping the slices. The function returns a value assigned to the two-dimensional context of the canvas each time you want to apply a gradient effect to it. After the function, let us pull out of the `for()` loop the definitions of the `sliceBorderStyle` and the `sliceBorderWidth` variables.

Listing 6-15. ch6_05.html

```
$(document).ready(function(){
    ...
    var dataSum = dataSum();
    var labels = $('<ul class="labels"></ul>')
       .insertBefore(canvas);

    function  sliceGradient(color){
       var sliceGradientColor = "#222";
       var sliceGradient = ctx.createLinearGradient(0, 0, w * .7, h * .7);
       sliceGradient.addColorStop(0, sliceGradientColor);
```

```
        sliceGradient.addColorStop(1, color);
        return sliceGradient;
    }
    var sliceBorderStyle = "#fff";
    var sliceBorderWidth = 4;

    for(var i in tableData.dataGroups){
        var sum = 0;
        var points = tableData.dataGroups[i];
        ...
    }
    ...
});
```

Another function you want to develop is `fraction()` (see Listing 6-16). This function calculates a value from 0 to 1 that represents the fraction of the slice in relation to the total pie. As you will see, the value returned will, in many cases, be useful.

Listing 6-16. ch6_05.html

```
$(document).ready(function(){
    ...
    var sliceBorderStyle = "#fff";
    var sliceBorderWidth = 4;

    function fraction(i) {
        var sum = 0;
        var points = tableData.dataGroups[i];
        for(var j in points){
            sum += points[j];
        }
        return (sum/dataSum);
    }

    for(var i in tableData.dataGroups){
        var sum = 0;
        var points = tableData.dataGroups[i];
        ...
    }
    ...
});
```

You now collect the whole sequence of functions, which you call with the context `ctx`, as demonstrated in Listing 6-17. This sequence, if correctly parameterized, is always the same, so you can collect it in a function you will call `drawSlice()`.

Listing 6-17. ch6_05.html

```
$(document).ready(function(){
    ...
    function fraction(i) {
        var sum = 0;
        var points = tableData.dataGroups[i];
```

```
        for(var j in points){
            sum += points[j];
        }
        return (sum/dataSum);
    }

    function drawSlice(centerx, centery, radius, counter, i) {
        var startAngle = counter * Math.PI * 2 - Math.PI * 0.5;
        var endAngle = (counter + fraction(i)) * Math.PI * 2 - Math.PI * 0.5;
        ctx.beginPath();
        ctx.moveTo(centerx, centery);
        ctx.arc(centerx, centery, radius, startAngle,endAngle, false);
        ctx.lineTo(centerx, centery);
        ctx.closePath();
        ctx.fillStyle = sliceGradient(colors[i]);
        ctx.fill();
        ctx.lineWidth = sliceBorderWidth;
        ctx.strokeStyle = sliceBorderStyle;
        ctx.stroke();
    }

    for(var i in tableData.dataGroups){
        var sum = 0;
        var points = tableData.dataGroups[i];
        ...
    }
    ...
});
```

You can apply the same logic to the code used to generate labels as an HTML unordered list <ul>, as shown in Listing 6-18.

Listing 6-18. ch6_05.html

```
$(document).ready(function(){
    ...
    function drawSlice(centerx,centery,radius,counter,i) {
        ...
        ctx.lineWidth = sliceBorderWidth;
        ctx.strokeStyle = sliceBorderStyle;
        ctx.stroke();
    }

    function drawLabels(i, counter) {
        var sliceMiddle = (counter + fraction(i)/2);
        var distance = radius * 1.2;
        var labelx = Math.round(centerx + Math.sin(sliceMiddle * Math.PI * 2) *
            (distance));
        var labely = Math.round(centery - Math.cos(sliceMiddle * Math.PI * 2) *
            (distance));
        var leftPlus = (labelx < centerx) ? '40' : '0' ;
        var percentage = parseFloat((fraction(i)*100).toFixed(2));
```

```
        var labelval = percentage + "%";
        var labeltext = $('<span class="label">' + labelval +'</span>')
           .css('font-size', radius / 8)
           .css('color', colors[i]);
        var label = $('<li class="label-pos"></li>')
           .appendTo(labels)
           .css({left: labelx-leftPlus, top: labely})
           .append(labeltext);
    }

    for(var i in tableData.dataGroups){
       var sum = 0;
       var points = tableData.dataGroups[i];
       ...
    }
    ...
});
```

After these statements, the part in the code handling the loop through the slices is reduced to what is shown in Listing 6-19.

Listing 6-19. ch6_05.html

```
$(document).ready(function(){
    ...
    function drawLabels(i, counter) {
    ...
    }

    for(var i in tableData.dataGroups){
       if(i == 2){
          counterAtI2 = counter;
       }else{
          drawSlice(centerx, centery, radius, counter, i);
       }
       drawLabels(i, counter);
       counter += fraction(i);
    }

    $('<div class="chart-title">'+table.find('caption').html() + '</div>')
       .insertBefore(canvas);
    ...
});
```

As you can see, the readability of the code has increased considerably, and it is also much easier to effect changes where necessary. After you have properly separated the various parts of the code, you move on to the code that handles the animation. The heart of the animation is the `setInterval()` function (see Listing 6-20). This function has two arguments: the first is a function that implements the action to be performed, and the second is the length of the time intervals between each execution of the function as first argument (in milliseconds). Hence, you define the `next()` function, which, for every execution, will draw the slice ever more distant from the center, until it reaches the

maxPullOutDistance. Therefore, you need a counter k, which will increase step by step until it reaches a maximum, in which each execution of the function draws the slice in the same place, making it appear as if the animation is finished (actually, it is always running).

Listing 6-20. ch6_05.html

```
$(document).ready(function(){
    ...
    for(var i in tableData.dataGroups){
       if(i == 2){
          counterAtI2 = counter;
       }else{
          drawSlice(centerx,centery,radius,counter,i);
       }
       drawLabels(i,counter);
       counter+=fraction(i);
    }

    var nextMove = setInterval(next, 100);
    var k = 0;
    function next() {
       var midAngle = Math.PI * (2 * counterAtI2 + fraction(2) - 0.5);
       var currentPullOutDistance = k;
       var maxPullOutDistance = 45;
       var ratio = currentPullOutDistance / maxPullOutDistance;
       var actualPullOutDistance = currentPullOutDistance * (Math.pow(1 - ratio, .8) + 1);
       var startx = centerx + Math.cos(midAngle) * actualPullOutDistance;
       var starty = centery + Math.sin(midAngle) * actualPullOutDistance;
       drawSlice(startx, starty, radius, counterAtI2, 2);
       if(k < maxPullOutDistance){
          k++;
       }else{
          k = maxPullOutDistance;
          clearInterval(nextMove);
       }
    }

    $('<div class="chart-title">'+table.find('caption').html() + '</div>')
       .insertBefore(canvas);
    ...
});
```

If you load the web page, you can see your animation, as presented left to right in Figure 6-7.

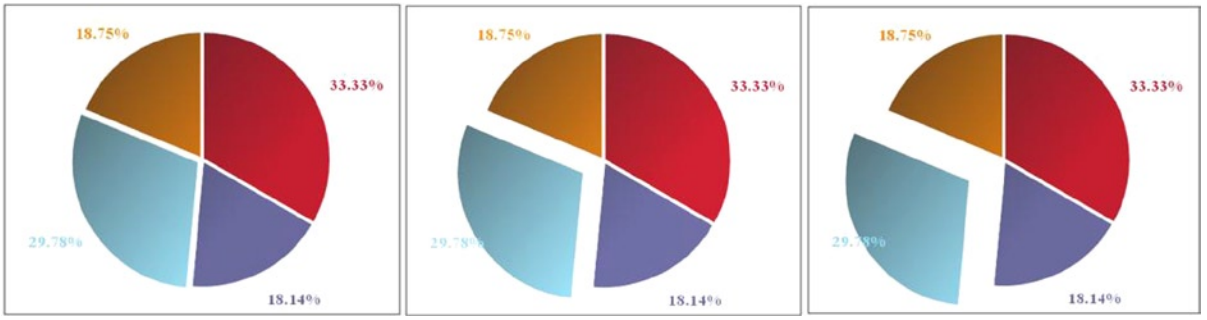


Figure 6-7. *Different frames of the animation while the slice is being pulled away from the pie chart*

Clicking a Slice to Pull It Out

In the previous example, you saw how you can implement an animation. In this and the following example you will take a further step: adding the ability to start the animation as a result of particular events. The most common type of event, and one that is well suited to pie charts, is to click a particular slice. . Once clicked, the slice will be pulled straight out of the pie. You will also learn to handle the return of the slice to the pie when another slice is clicked.

You write a new `slice()` function, which will build a slice object and all its attributes, such as `startAngle`, `endAngle`, `counter`, and `fraction` (see Listing 6-21). So far, you have done this implicitly, but this way, is more correct. You also define the `sliceAll` array, which will contain all the slices of the pie.

Listing 6-21. ch6_06.html

```
$(document).ready(function(){
    ...
    function drawLabels(i, counter) {
       ...
    }

    function slice(counter,i){
       var startAngle = counter * Math.PI * 2 - Math.PI * 0.5;
       var endAngle = (counter + fraction(i)) * Math.PI * 2 - Math.PI * 0.5;
       this.startAngle = startAngle;
       this.endAngle = endAngle;
       this.counter = counter;
       this.fraction = fraction(i);
       return this;
    }
    var allSlices = new Array();

    for(var i in tableData.dataGroups){
       ...
    }
    ...
});
```

You modify the `for()` loop slightly just by adding the definition of slices, using the new `slice()` function, as shown in Listing 6-22.

Listing 6-22. ch6_06.html

```
$(document).ready(function(){
    ...
    var allSlices = new Array();

    for(var i in tableData.dataGroups){
       allSlices[i] = new slice(counter, i);
       drawSlice(centerx, centery, radius, counter, i);
       drawLabels(i, counter);
       counter += fraction(i);
    }
    var nextMove = setInterval(next, 100);
    var k = 0;
    ...
});
```

Before going any further, let us add the global variables to the code (see Listing 6-23). At the same time, the `nextMove` and `k` variables must to be deleted or commented out.

Listing 6-23. ch6_06.html

```
$(document).ready(function(){
    ...
    for(var i in tableData.dataGroups){
       allSlices[i] = new slice(counter,i);
       drawSlice(centerx,centery,radius,counter,i);
       drawLabels(i,counter);
       counter+=fraction(i);
    }
    var sliceToPullout = -1;
    var sliceToPullin  = 0;
    //var nextMove = setInterval(next, 100);
    //var k = 0;
    function next() {
       ...
    }
    ...
});
```

Next, you activate the capture of the mouse `click` event on the canvas and link it to a function you will call `handleChartClick()`, as illustrated in Listing 6-24.

Listing 6-24. ch6_06.html

```
$(document).ready(function(){
    ...
    function next() {
       ...
    }
```

```
    $('#myCanvas').click(handleChartClick);
    $('<div class="chart-title">'+table.find('caption').html() + '</div>')
       .insertBefore(canvas);
    $('table').hide();
});
```

Now, you have to implement the`handleChartClick()` function, which you have just inserted as argument in the `click()` function (see Listing 6-25). The `mouseX` and `mouseY` variables store the coordinates of the point on the canvas where you click the mouse button. These values are analyzed to ascertain whether they correspond to the pie surface and to which slice they correspond. Once the slice has been identified, it is marked as outgoing, and the one currently pulled out is marked as incoming. Then, the `next()` function is thrown. This implements the actions to be taken on the canvas.

Furthermore, you must take into consideration the possibility that the clicked slice is the slice extracted. In this case, you go back to the initial state, when all the slices are in the pie.

Listing 6-25. ch6_06.html

```
$(document).ready(function(){
    ...
    function next() {
       ...
    }

    function handleChartClick ( clickEvent ) {
       var mouseX = clickEvent.pageX - this.offsetLeft;
       var mouseY = clickEvent.pageY - this.offsetTop;
       var xFromCentre = mouseX - centerx;
       var yFromCentre = mouseY - centery;
       var distanceFromCentre =
          Math.sqrt( Math.pow( Math.abs( xFromCentre ), 2 ) +
          Math.pow( Math.abs( yFromCentre ), 2 ) );
       if ( distanceFromCentre <= radius ) {
          var clickAngle = Math.atan2( yFromCentre, xFromCentre );
          if(yFromCentre < 0 && xFromCentre < 0)
             clickAngle = (Math.PI + clickAngle) + Math.PI;
          for ( var i in allSlices ) {
             if ( clickAngle >= allSlices[i].startAngle &&
                      clickAngle <= allSlices[i].endAngle ) {
                sliceToPullin = sliceToPullout;
                sliceToPullout = i;
                if(sliceToPullout == sliceToPullin)
                sliceToPullout = -1;
                next(sliceToPullout,sliceToPullin);
             }
          }
       }
    }
    $('#myCanvas').click ( handleChartClick );
    $('<div class="chart-title">'+table.find('caption').html() + '</div>')
       .insertBefore(canvas);
    $('table').hide();
});
```

You replace the already existing `next()` function with a new one that takes two parameters: `out` and `ins` (see Listing 6-26). Each slice is identified with a number that corresponds to the index of the loop through the series. `Out` and `ins` are variables that store information about which slice must be pulled out of the pie and which slice must be reinserted. Only when a slice is clicked is the event captured, and the `next()` function is thrown. The first thing it does is to clear the entire canvas with the `clearRect()` function. As the loop is passed through all the slices, they are drawn, one by one. If the slice's index is equal to the variable `out`, the slice will be pulled out of the pie, but if its index is equal to `ins`, it will be pulled into the pie.

Listing 6-26. ch6_06.html

```
$(document).ready(function(){
    ...
    function next(out,ins) {
       ctx.clearRect ( 0, 0, canvas.width(), canvas.height() );
       ctx.lineWidth = 1;
       ctx.strokeStyle = '#000';
       ctx.strokeRect(margin.left,margin.top,w,h);
       for(var i in allSlices){
          var counter = allSlices[i].counter;
          var startAngle = allSlices[i].startAngle;
          var endAngle = allSlices[i].endAngle;
          var fraction = allSlices[i].fraction;
          var maxPullOutDistance = 30;
          if( i == out){
          //Pull out
             var currentPullOutDistance = 30;
             var ratio = currentPullOutDistance / maxPullOutDistance;
             var midAngle = (startAngle + endAngle) / 2;
             var actualPullOutDistance =
             currentPullOutDistance * (Math.pow( 1 - ratio, .8 ) + 1);
             var startx = centerx + Math.cos(midAngle) * actualPullOutDistance;
             var starty = centery + Math.sin(midAngle) * actualPullOutDistance;
             drawSlice(startx, starty, radius, counter, i);
          }else if(i == ins){
          //Push In
             var currentPullOutDistance = 0;
             var ratio = currentPullOutDistance / maxPullOutDistance;
             var midAngle = (startAngle + endAngle) / 2;
             var actualPullOutDistance = currentPullOutDistance *
                 (Math.pow( 1 - ratio, .8 ) + 1);
             var startx = centerx + Math.cos(midAngle) * actualPullOutDistance;
             var starty = centery + Math.sin(midAngle) * actualPullOutDistance;
             drawSlice(startx, starty, radius, counter, i);
          }else{
             drawSlice(centerx, centery, radius, counter, i);
          }
       }
    }
```

```
    function handleChartClick ( clickEvent ) {
       ...
    }
    ...
});
```

Now, when you load the web page, you get an interactive pie chart (see Figure 6-8)!

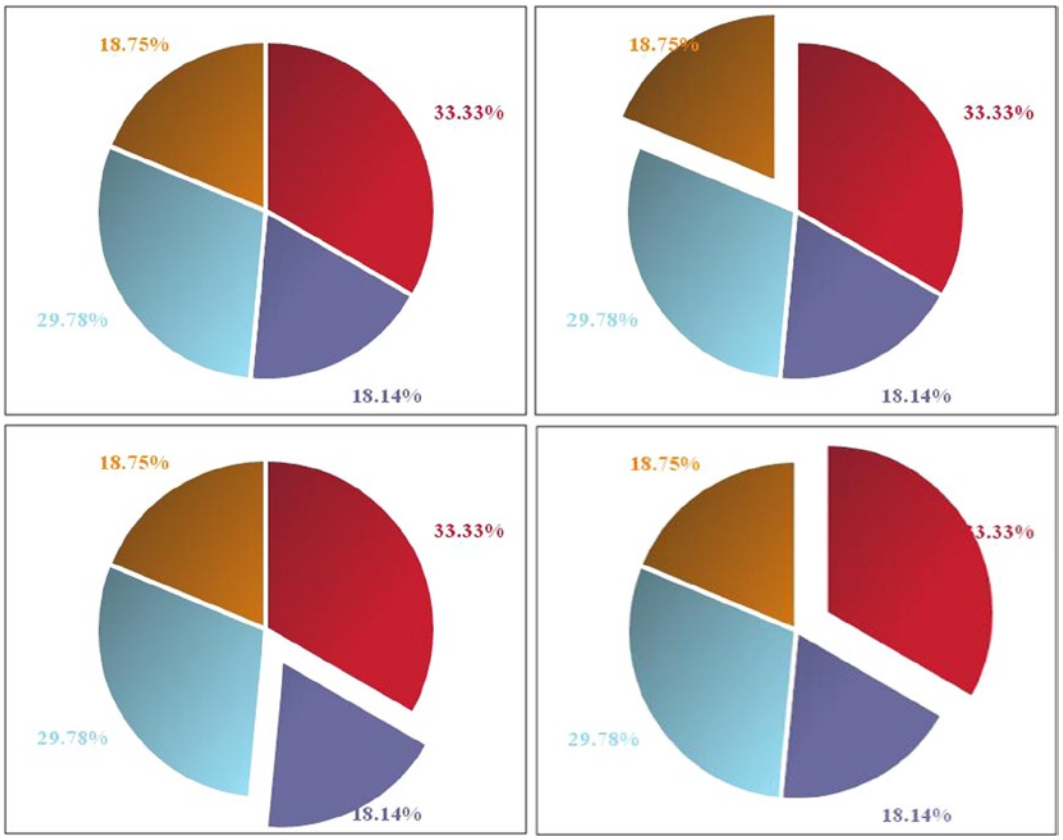


Figure 6-8. *The user can choose which slice to extract from the pie*

Clicking a Slice to Pull It Out with Animation

Now that you understand how to connect an event to a function, you have to make sure that the mouse click activates an animation. First, you add two counters, as shown in Listing 6-27. These are used to store the steps (distance) to cover from the center of the pie for the outgoing and incoming slices. When one counter is at the maximum distance, the other must be at 0.

Listing 6-27. ch6_06.html

```
$(document).ready(function(){
    ...
    var sliceToPullout = -1;
    var sliceToPullin  = 0;
    var k1 = 0;
    var k2 = 20;
    function next() {
       ...
    }
    ...
});
```

You insert two animations simultaneously, one handling the incoming slice and the other handling the outgoing slice. To accomplish this, you call two different `setInterval()` functions; one for when the index `i` matches the `out` value and the other for when `i` matches the `ins` value (see Listing 6-28). You must also define two different functions that describe two different actions so that you can pass them as arguments to the two `setInterval()` functions.

Listing 6-28. ch6_06.html

```
$(document).ready(function(){
    ...
    function next(out,ins) {
       ctx.clearRect ( 0, 0, canvas.width(), canvas.height() );
       ctx.lineWidth = 1;
       ctx.strokeStyle = '#000';
       ctx.strokeRect(margin.left,margin.top,w,h);
       for(var i in allSlices){
          var counter = allSlices[i].counter;
          var startAngle = allSlices[i].startAngle;
          var endAngle = allSlices[i].endAngle;
          var fraction = allSlices[i].fraction;
          var maxPullOutDistance = 25;
          if( i == out){
             var nextMove = setInterval(pullOut, 100);
          }else if(i == ins){
             var nextMove = setInterval(pushIn, 100);
          }else{
             drawSlice(centerx, centery, radius, counter, i);
          }
    }
}

    function handleChartClick ( clickEvent ) {
       ...
    }
    ...
});
```

Now, you have to implement the `pullout()` function (see Listing 6-29). There is nothing new here, compared with the previous examples, except that at the end, you need to manage the `k1` counter.

Listing 6-29. ch6_06.html

```
$(document).ready(function(){
    ...
    function next() {
       ...
    }

    function pullOut(){
       var s = sliceToPullout;
       var counter = allSlices[s].counter;
       var startAngle = allSlices[s].startAngle;
       var endAngle = allSlices[s].endAngle;
       var fraction = allSlices[s].fraction;
       var maxPullOutDistance = 25;
       var currentPullOutDistance = k1;
       var ratio = currentPullOutDistance / maxPullOutDistance;
       var midAngle = (startAngle + endAngle) / 2;
       var actualPullOutDistance = currentPullOutDistance * (Math.pow( 1 - ratio, .8 ) + 1);
       var startx = centerx + Math.cos(midAngle) * actualPullOutDistance;
       var starty = centery + Math.sin(midAngle) * actualPullOutDistance;
       drawSlice(startx, starty, radius, counter, s);
       if(k1 < 20){
          k1++;
       }else{
          k1 = 20;
          clearInterval(nextMove);
       }
    }

    function handleChartClick ( clickEvent ) {
       ...
    }
    ...
});
```

The same applies for the `pushIn()` function, as demonstrated in Listing 6-30.

Listing 6-30. ch6_06.html

```
$(document).ready(function(){
    ...
    function pullOut() {
       ...
    }

    function pushIn(){
       var s = sliceToPullin;
       var counter = allSlices[s].counter;
       var startAngle = allSlices[s].startAngle;
       var endAngle = allSlices[s].endAngle;
       var fraction = allSlices[s].fraction;
       var maxPullOutDistance = 25;
```

```
        var currentPullOutDistance = k2;
        var ratio = currentPullOutDistance / maxPullOutDistance;
        var midAngle = (startAngle + endAngle) / 2;
        var actualPullOutDistance = currentPullOutDistance * (Math.pow( 1 - ratio, .8 ) + 1);
        var startx = centerx + Math.cos(midAngle) * actualPullOutDistance;
        var starty = centery + Math.sin(midAngle) * actualPullOutDistance;
        drawSlice(startx, starty, radius, counter, s);
        if(k2 > 0){
           k2--;
        }else{
           k2 = 0;
           clearInterval(nextMove);
        }
     }

     function handleChartClick ( clickEvent ) {
        ...
     }
     ...
});
```

For the event handler function, `handleChartClick()`, you need to reset the values for the `k1` and `k2` counters so that a new animation starts every time you click a slice (see Listing 6-31).

Listing 6-31. ch6_06.html

```
$(document).ready(function(){
    ...
    function handleChartClick ( clickEvent ) {
       ...
       if ( distanceFromCentre <= radius ) {
          var clickAngle = Math.atan2( yFromCentre, xFromCentre );
          if(yFromCentre < 0 && xFromCentre < 0)
             clickAngle = (Math.PI + clickAngle) + Math.PI;
          for ( var i in allSlices ) {
             if ( clickAngle >= allSlices[i].startAngle &&
                    clickAngle <= allSlices[i].endAngle ) {
                sliceToPullin = sliceToPullout;
                sliceToPullout = i;
                if(sliceToPullout == sliceToPullin)
                   sliceToPullout = -1;
                k1 = 0;
                k2 = 20;
                next(sliceToPullout,sliceToPullin);
             }
          }
       }
    }
    ...
});
```

With this example, you can see the possibility of implementing a chart with interactive animations that react to events triggered by the users.

Other Effects

Throughout the book, you will find that there are other effects that you can add to your chart, and you will see many of them are already implemented in specialized libraries in the representation of charts.

One such effect drawing a chart on the browser, can be done so that the elements that compose the chart are drawn one after the other, instead of all at once. This effect results in a fluctuating animation. Another effect that will be discussed in detail is the highlighting of the elements representing data (such as a slice, a bar, or a data point on a line). When the user mouses over one of these items, the chart may show some small animations, such as changing the shape or color of the element or creating a small box containing additional information (a tooltip).

Summary

With this chapter, you have finished learning about the development of the three most common charts, using the same set of data obtained from an HTML table. You have also seen that it is possible to enrich your chart by inserting animations in response to certain events triggered by the user, such as clicking one of the slice in **a pie chart** to pull it out of the chart.

In the next chapter, you will come to the end of the first part of this book. You will discover how all that you have implemented separately in the last few chapters can be merged into a single module: a **JavaScript library** that produces charts. You will come to understand how these kinds of libraries work—libraries that you will study in detail in other parts of the book.

CHAPTER 7

Creating a Library for Simple Charts

As a conclusion to this first part of the book, you will use everything you have learned so far to create a library of your own from scratch. This will be a library specialized in representing the three different types of charts that you have seen so far: line, bar, and pie charts.

What you will be developing is a very simplified model of the JavaScript libraries currently available on the Internet. The purpose is to help you understand the mechanisms that underlie such specialized libraries for chart representation.

By including all the steps in this simple example, you can better see how this class of libraries works, even for examples that are much more complex. Following the flow of data, from their definition in the HTML page to their processing within the library, you will discover how the data are converted into graphic elements to form the type of chart you are most interested in. The gradual implementation of this library will illuminate the reasons why the jQuery library is the basis of many such libraries. Thanks to its functions, it is possible to manage the components of the HTML page dynamically. This library also plays a key role in the implementation and management of a number of parameters that will have a direct influence on the properties of graphic elements created, thus characterizing the different chart representations that are possible without your having to modify the code, each time specifying a JavaScript object in which you will pass all these parameters.

In the previous chapters, you have already developed the code you will need here. You have seen how to manage data iteratively, how to convert the data into graphics, thanks to the context of the canvas and the many functions that jQuery provides. In this way, you created three of the most common types of charts.You will use the code that you developed to create your library, discovering how to parameterize it so that you can decide which chart to represent and how to do so when it is time to call your library, without modifying the code.

Creating a Library

First, you need to define your new library and include it in your web page. To this end, you create a new file that contains the definition of the `myLibrary()` function. You save this file as `myLibrary.js`; this will be your library and can be reused whenever you want to include it in a web page (see Listing 7-1).

Listing 7-1. myLibrary.js

```
function myLibrary(target, data, options){
    //add the JavaScript code here
}
```

At the same time, you begin to implement a new web page, including the `myLibrary` file in it, as shown in Listing 7-2.

Listing 7-2. ch7_01.html

```
<HTML>
<HEAD>
<TITLE>MyChart</TITLE>
</HEAD>
<BODY>
<script type="text/javascript" src="../src/jquery.min.js"></script>
<script type="text/javascript" src="./mylibrary.js"></script>
<script>
$(document).ready(function(){

    // add data and options here

    myLibrary("#myCanvas", data, options);
});
</script>
<canvas id="myCanvas" width="500" height="400"> </canvas>
</BODY>
</HTML>
```

As you can see, first you have included the jQuery library in the web page, so that you will be able to take advantage of all the methods that this library offers when using your code.

■ **Note** If you prefer to include the jQuery library with a content delivery network (CDN) service, you need to use this reference:

```
<script src="http://code.jquery.com/jquery-1.9.1.min.js"></script>
```

For further information about the workspace and library's path, see Appendix A.

Main Features: Target, Data, and Options

Inside the `$(document).ready()` function, you call your library with the `myLibrary()` function. With this call, you pass three different arguments:

- `target`
- `data`
- `options`

You will find that this type of call is very common in many libraries, including jqPlot and Highcharts. Thus, in implementing your library, you already begin to deal with the concepts that will form the basis of subsequent chapters.

target is the name given to the ID class of the canvas. The `target` is passed into the library to enable you to define a context and draw all the graphics needed inside. Nothing prevents you from using different canvases in the same web page, each representing a different type of chart, but all must be distinguishable by means of a different name given to the `target`.

data is the array containing your input data. In the previous chapters, you used the data contained in a table and pulled it out through the parser you implemented. This helped you understand the potential of jQuery, but, in fact, most of the time these input values can have any origin and assume any form. Usually, the conversion into a readable format is not the job of the library, but of other, supporting code. Thus, for your library, the input format must be an array.

options is an object data type and can also assume complex structures, for which you need to specify a number of properties associated with attribute values. You will use this type of structure to pass a whole series of parameters to the library, characterizing all the graphic components of your chart. Basically, this entails defining a set of guidelines on how you want your library to represent your chart.

Once you become familiar with these basic concepts, you will see how all the libraries covered by this book, as well as others on the Internet, will be much easier to comprehend and use.

The library that you are implementing, like all others, will accept input data in the form of an array. Instead of implementing the parser, which extracts values from an HTML table, you write data directly, in the form of numerical arrays. The `data` variable will be defined within `$(document).ready()` before calling the `myLibrary()`function (see Listing 7-3).

Listing 7-3. ch7_01.html

```
$(document).ready(function(){
    var data = [[12, 40, 75, 23, 42, 80],
                [3, 22, 40, 27, 35, 21],
                [60, 80, 16, 28, 33, 26],
                [46, 7, 14, 26, 36, 24]];
    myLibrary("#myCanvas",data,options);
});
```

Asmentioned earlier, these are the numerical values of the HTML table (see Figure 7-1). But, what happened to the heading, with the months of the year and the names of nations?You will express these two sets of values as an array as well, but instead of introducing them as input data, you use them as properties of the chart to be inserted through `options`.

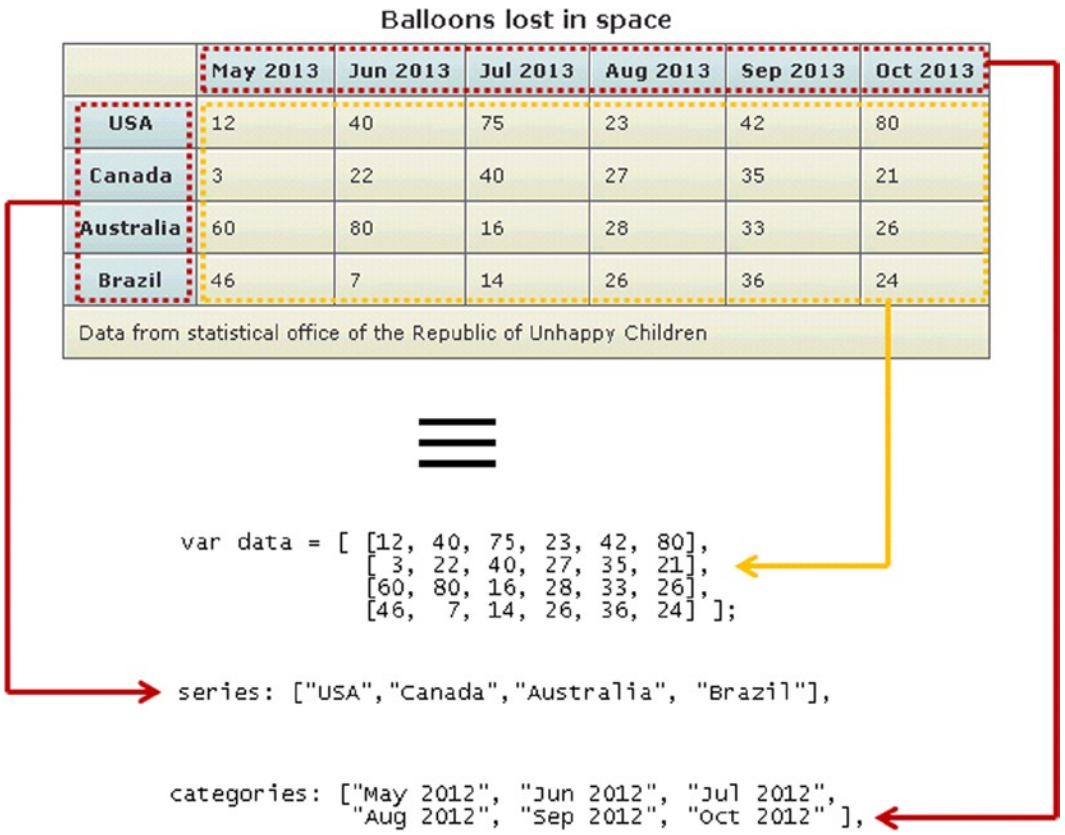


Figure 7-1. *The input data can come from any kind of source (e.g., a table); the important thing to keep the data structure*

You can consider the months of the year tick labels (also called **categories**), but with regard to the names of nations, these are nothing more than the names of the **series** of values, which will be reported in a legend and to which a different color will be assigned. In fact, most of the labelswill be assigned to the components of the chart, so it is preferable to pass them through `options` (see Listing 7-4). Previously, you defined a sequence of **colors** through an array. Thus, you will pass this array in `options` as well.

Listing 7-4. ch7_01.html

```
$(document).ready(function(){
    var data = [[12, 40, 75, 23, 42, 80],
                [3, 22, 40, 27, 35, 21],
                [60, 80, 16, 28, 33, 26],
                [46, 7, 14, 26, 36, 24]];
    var options = {
       categories: ["May 2012", "Jun 2012", "Jul 2012",
                    "Aug 2012", "Sep 2012", "Oct 2012"],
       series: ["USA","Canada","Australia", "Brazil"],
       colors: ['#be1e2d', '#666699', '#92d5ea', '#ee8310'],
    };
    myLibrary("#myCanvas",data,options);
});
```

However, you have forgotten the most important thing. What type of chart do you want to use? You will specify this information in `options` as well, defining, in this particular case, a `type` property with three possible values:

- `line`
- `bar`
- `pie`

This is the tip of the iceberg in terms of what you can parameterize within your chart. In implementing your library, any parameter that characterizes the appearance or functionality of a graphic element can be set externally through the `options` object, as demonstrated in Listing 7-5.

Listing 7-5. ch7_01.html

```
$(document).ready(function(){
    var data = [[12, 40, 75, 23, 42, 80],
                [3, 22, 40, 27, 35, 21],
                [60, 80, 16, 28, 33, 26],
                [46, 7, 14, 26, 36, 24]];
    var options = {
       //type: 'line',
       type: 'bar',
       //type: 'pie',
       categories: ["May 2012", "Jun 2012", "Jul 2012",
                    "Aug 2012", "Sep 2012", "Oct 2012"],
       series: ["USA","Canada","Australia", "Brazil"],
       colors: ['#be1e2d', '#666699', '#92d5ea', '#ee8310'],
    };
    myLibrary("#myCanvas",data,options);
});
```

Here, we take only a few, small cases as examples, because our purpose is illustrative; what is important is to understand the basic methodology. For instance, when defining the canvas in previous cases, you specified margins in the drawing area. However, this is a case in which it would be more appropriate to leave the possibility open for these parameters to be defined outside the library, directly by the users.

In other cases, there might be even more specific parameters, typical of a single type of chart. In this case, you will have a further nested structure, such as an `options` object inside another `options` object that is specific to only one type of chart. Such an approach is the basis, for example, of the large number of properties and subproperties that constitute the `options` object in jqPlot, a library that you will look into in the next part of the book. Therefore, by way of example, let us insert the `barGroupMargin` property as a specific parameter to the bar chart (see Figure 7-2). With this property, you can control the distance between the bars. Because this property is specific to only one type of chart, it will be specified within a `bar` object, contained in turn inside `options`.

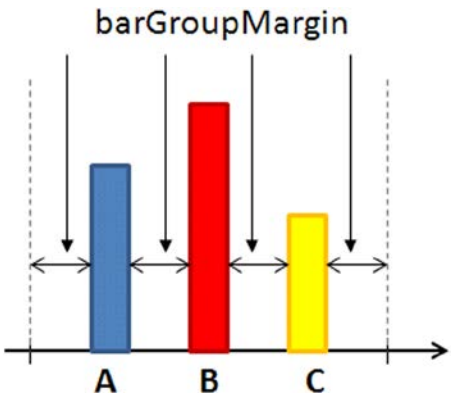


Figure 7-2. *Setting the* `barGroupMargin` *property, you can modify the distance between the bars*

Even the margins of the canvas can be defined as properties within `options`. In this way, you can adjust the position of your chart without needing each time to change the code in the `myLibrary.js` library.

Using this approach, you subdivide the properties, depending on the area of influence in the hierarchy of `options`, assigning them to the object that describes that area (see Figure 7-3).

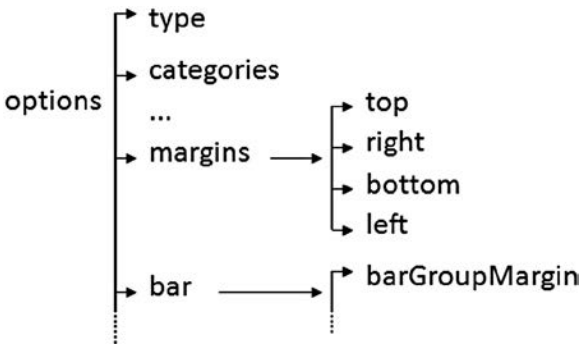


Figure 7-3. *The hierarchy of the* `options` *object reflects the hierarchy of the elements that form the chart*

In this case, you have a number of properties to be set within the `options` object, as shown in Listing 7-6.

Listing 7-6. ch7_01.html

```
$(document).ready(function(){
    var data = [[12, 40, 75, 23, 42, 80],
                [3, 22, 40, 27, 35, 21],
                [60, 80, 16, 28, 33, 26],
                [46, 7, 14, 26, 36, 24]];
    var options = {
       //type: 'line',
       type: 'bar',
       //type: 'pie',
       categories: ["May 2012", "Jun 2012", "Jul 2012",
                    "Aug 2012", "Sep 2012", "Oct 2012"],
```

```
        series: ["USA","Canada","Australia", "Brazil"],
        colors: ['#be1e2d', '#666699', '#92d5ea', '#ee8310'],
        margins: {top: 30, right: 10, bottom: 10, left: 30},
        bar: {
           barGroupMargin: 4
        }
    };
    myLibrary("#myCanvas",data,options);
});
```

You have thus finished defining all there is to define in the web page. Now, the input data must be processed and converted into graphic elements, using the `data` array. You also have a whole series of parameters characterizing the chart within your `options` object. Finally, you have indicated the `target`, that is, the canvas on which you will draw your chart. So, let us look inside `myLibrary` to tackle all these issues.

Implementing the Library

Now that you have finished defining everything in the web page, you must begin to implement your library. If you go back to look at the code that you used to obtain your line chart, bar chart, and pie chart, you will find that these codes have many parts in common. It is these common parts that form the backbone of the library, whereas those parts specific to a type of chart will be implemented separately, within an `if()` statement that will activate only if the type selected in `options` corresponds.

Setting the Canvas

One of the common parts of the code is the definition of the context to apply to the canvas, shown in Listing 7-7.

Listing 7-7. myLibrary.js

```
function myLibrary(target,data,options){
    var canvas = $(target);
    var margin = options.margins;
    var w = canvas.width() - margin.left - margin.right,
        h = canvas.height() - margin.top - margin.bottom;

    var ctx = canvas.get(0).getContext("2d");
    if(options.type === 'pie'){
       ctx.strokeRect(margin.left, margin.top, w, h);
    } else {
       ctx.translate( 0, canvas.height() );
       ctx.strokeRect(margin.left, -margin.bottom, w, -h);
    }
};
```

As you can see, it is here that the `target` argument is used. With regard to the definition of the margins, you must remember to define them within the `options` object, and so you will read the values that have been defined inside.To access these values is really very simple; you just have to define the statement each time it is necessary:

`options.`*`property`*

Thus, margins will use `options.margins`. In this part of code, the only thing that distinguishes between the three types of charts is the rectangle defining the design area, the orientation of which agrees with the page for the pie chart, whereas for the line chart and the bar chart, the orientation is reversed. That is why `ctx.translate()` is applied only if `options.type` is different from `'pie'`.

Drawing the Axes, Tick Labels, and Grid

Now, let us add the code to your library. This code handles the creation of the x and y axis tick labels. These components are only needed for line charts and bar charts, as these are represented on the Cartesian axes; the pie chart does not use them. Therefore, you apply the condition inside the `if()` statement so that the code is executed only for these two types of charts. The code you are implementing is the same as that used in the previous chapters, except that in this case, you have replaced the variables with others available in the `options` object and `data` array (see Listing 7-8).

Listing 7-8. myLibrary.js

```
function myLibrary(target,data,options){
    ...
    var ctx = canvas.get(0).getContext("2d");
    if(options.type === 'pie'){
       ctx.strokeRect(margin.left, margin.top, w, h);
    } else {
       ctx.translate( 0, canvas.height() );
       ctx.strokeRect(margin.left, -margin.bottom, w, -h);
    }

    if(options.type === 'line' || options.type === 'bar'){
       var minVal = 0;
       var maxVal = 0;
       data.forEach(function(d,i){
          var min = Math.min.apply(null, d);
          if(min < minVal)
             minVal = min;
          var max = Math.max.apply(null, d);
          if(max > maxVal)
             maxVal = max;
       });
       maxVal = 1.1 * maxVal;

       //calculate yLabels
       var yLabels = [];
       var yDeltaPixels = 30;
       var nTicks = Math.round(h / yDeltaPixels);
       var yRange = maxVal - minVal;
       var yDelta = Math.ceil(yRange / nTicks);
       var yVal = minVal;
       while(yVal < (maxVal - yDelta)){
          yLabels.push(yVal);
          yVal += yDelta;
       }
       yLabels.push(yVal);
       yLabels.push(maxVal);
```

```
        //draw xLabels
        if(options.type === 'line'){
           var xDelta = w / (options.categories.length - 1);
        }
        if(options.type === 'bar'){
           var xDelta = w / (options.categories.length);
        }
        var xlabelsUL = $('<ul class="labels-x"></ul>')
           .width(w)
           .height(h)
           .insertBefore(canvas);
        $.each(options.categories, function(i){
           var thisLi = $('<li><span class="label">' + this + '</span></li>')
              .prepend('<span class="line" />')
              .css('left', xDelta * i)
              .width(0)
              .appendTo(xlabelsUL);
           var label = thisLi.find('span.label');
        });

        //draw yLabels
        var yScale = h / yRange;
        var liBottom = h / (yLabels.length-1);
        var ylabelsUL = $('<ul class="labels-y"></ul>')
           .width(w)
           .height(h)
           .insertBefore(canvas);
        $.each(yLabels, function(i){
           var thisLi = $('<li><span>' + this + '</span></li>')
              .prepend('<span class="line"  />')
              .css('bottom', liBottom * i)
              .prependTo(ylabelsUL);
           var label = thisLi.find('span:not(.line)');
           var topOffset = label.height()/-2;
           if(i == 0){ topOffset = -label.height(); }
           else if(i== yLabels.length-1){ topOffset = 0; }
           label
              .css('margin-top', topOffset)
              .addClass('label');
        });
     }
};
```

If you take a look at the Cascading Style Sheets (CSS) styles defined in the previous chapters, for the three types of charts, you will see that they are not all the same, especially for certain classes of style. To overcome this problem, the simplest approach is to define these attributes, using the `css()` function for the various classes of style (or, rather, the tag representing them) at the time of their definition. Thus, you can have style classes with the same name for all three types of charts, but with different values precisely because each of them has its own `css()` functions.

For example, when you define the CSS class `span.label`, which regulates the x axis tick labels, these must behave differently, according to whether you are working on a line chart or a bar chart. If you want to represent a line chart, the tick labels will be reported at the ticks, but if you want to represent a bar chart, the labels should be reported at

two ticks. You therefore have to define the attributes of the same `span.label` class in a different way, and you do so by adding a left margin of 40 pixels with the `css()` function exclusive to bar charts. The relevant part of the code is shown in Listing 7-9.

Listing 7-9. myLibrary.js

```
function myLibrary(target,data,options){
    ...
    if(options.type === 'line' || options.type === 'bar'){
       ...
       $.each(options.categories, function(i){
          var thisLi = $('<li><span class="label">' + this + '</span></li>')
             .prepend('<span class="line" />')
             .css('left', xDelta * i)
             .width(0)
             .appendTo(xlabelsUL);
          var label = thisLi.find('span.label');
          if(options.type === 'line'){
             label.addClass('label');
          }
          if(options.type === 'bar'){
             label.css('margin-left', '40px')
                .addClass('label');
          }
       });

       //draw yLabels
       var yScale = h / yRange;
       var liBottom = h / (yLabels.length-1);
       ...
    }
};
```

Because we are talking about CSS classes, let us add to your web page all the definitions that you used in the previous chapters, definitions that remain valid for all three types. But, instead of adding them directly to your web page, writing them between the `<style></style>` tags, you have to considerthese CSS definitions a part of the library; therefore, it is better to write them in a new CSS file, which you will call `myLibrary.css` (see Listing 7-10).

Listing 7-10. myLibrary.css

```
canvas {
     position: relative;
}
ul,.li {
    margin: 0;
    padding: 0;
}
.labels-x, .labels-y {
    position: absolute;
    left: 37;
    top: 37;
    list-style: none;
}
```

```
.labels-x li {
    position: absolute;
    bottom: 0;
    height: 100%;
}
.labels-x li span.label {
    position: absolute;
    color: #555;
    top: 100%;
    margin-top: 5px;
    left:-15;
}
.labels-x li span.line{
    position: absolute;
    border: 0 solid #ccc;
    border-left-width: 1px;
    height: 100%;
}
.labels-y li {
    position: absolute;
    bottom: 0;
    width: 100%;
}
.labels-y li span.label {
    position: absolute;
    color: #555;
    right: 100%;
    margin-right: 5px;
    width: 100px;
    text-align: right;
}
.labels-y li span.line {
    position: absolute;
    border: 0 solid #ccc;
    border-top-width: 1px;
    width: 100%;
}
.legend {
    list-style: none;
    position: absolute;
    left: 520px;
    top: 40px;
    border: 1px solid #000;
    padding: 10px;
}
.legend li span {
    width: 12px;
    height: 12px;
    float: left;
    margin: 3px;
}
```

```
.chart-title {
    font-size: 24;
    font-weight: bold;
    position: absolute;
    left: 150px;
    top: 10px;
    width: 100%
}
```

To make these CSS style settings effective, the new CSS file must be included in your web page, along with a link to the file, as presented in Listing 7-11.

Listing 7-11. myLibrary.js

```
<HEAD>
<TITLE>MyChart</TITLE>
</HEAD>
<BODY>
<script type="text/javascript" src="../src/jquery.min.js"></script>
<link href="./myLibrary.css" rel="stylesheet" type="text/css">
<script type="text/javascript" src="./mylibrary.js"></script>
<script>
...
```

Drawing Data

You now have to define the portion of the code that converts the input data into graphic elements, using the context of the canvas (see Listing 7-12). This part is specific to each type of chart, and so you will have a different implementation for each.

Listing 7-12. myLibrary.js

```
function myLibrary(target,data,options){
    ...
    if(options.type === 'line' || options.type === 'bar'){
       ...
    }
    if(options.type === 'line'){
       // draw DATA
       ctx.lineWidth = 5;
       for(var i in data){
          var points = data[i];
          ctx.moveTo(0,-points[i]);
          ctx.strokeStyle = options.colors[i];
          ctx.beginPath();
          var xVal = margin.left;
          for(var j in points){
             var relY = (points[j] * h / maxVal) + 10;
             ctx.lineTo(xVal,-relY);
             xVal += xDelta;
          }
```

```
      ctx.stroke();
      ctx.closePath();
   }
} // end of LINE

if(options.type === 'bar'){
   var barGroupMargin = options.bar.barGroupMargin;
   for(var i in data){
      ctx.beginPath();
      var n = data.length;
      var lineWidth = (xDelta - barGroupMargin * 2) / n;
      var strokeWidth = lineWidth - (barGroupMargin * 2);
         ctx.lineWidth = strokeWidth;
      var points = data[i];
      var xVal = (xDelta - n * strokeWidth - (n - 1) * (lineWidth - strokeWidth)) / 2;
      for(var j in points){
         var relX = margin.left + (xVal - barGroupMargin) +
            (i * lineWidth) + lineWidth / 2;
         ctx.moveTo(relX,-margin.bottom);
         var relY = margin.bottom + points[j] * h / maxVal;
         ctx.lineTo(relX, -relY);
         xVal += xDelta;
      }
      ctx.strokeStyle = options.colors[i];
      ctx.stroke();
      ctx.closePath();
   }
} // end of bar

if(options.type === 'pie'){
   var pieMargin = margin.top + 50;
   var centerx = Math.round(w / 2) + margin.left;
   var centery = Math.round(h / 2) + margin.top;
   var radius = centery - pieMargin;
   var counter = 0.0;
   var dataSum = function(){
      var dataSum = 0;
      for(var i in data){
         var points = data[i];
         for(var j in points){
            dataSum += points[j];
         }
      }
      return dataSum;
   }
   var dataSum = dataSum();
   var labels = $('<ul class="labels"></ul>')
         .css('list-style','none')
      .insertBefore(canvas);
   for(var i in data){
      var sum = 0;
```

```
            var points = data[i];
            for(var j in points){
               sum += points[j];
            }
            var fraction = sum / dataSum;
            ctx.beginPath();
            ctx.moveTo(centerx, centery);
            ctx.arc(centerx, centery, radius,
               counter * Math.PI * 2 - Math.PI * 0.5,
               (counter + fraction) * Math.PI * 2 - Math.PI * 0.5, false);
            ctx.lineTo(centerx, centery);
            ctx.closePath();
            ctx.fillStyle = options.colors[i];
            ctx.fill();
            var sliceMiddle = (counter + fraction / 2);
            var distance = radius * 1.2;
            var labelx = Math.round(centerx +
               Math.sin(sliceMiddle * Math.PI * 2) * (distance));
            var labely = Math.round(centery -
               Math.cos(sliceMiddle * Math.PI * 2) * (distance));
            var leftPlus = (labelx < centerx) ? '40' : '0' ;
            var percentage = parseFloat((fraction * 100).toFixed(2));
            var labelval = percentage + "%";
            var labeltext = $('<span class="label">' + labelval +'</span>')
               .css('font-size', radius / 8)
               .css('color', options.colors[i])
               .css('font-weight', 'bold');
            var label = $('<li class="label-pos"></li>')
               .appendTo(labels)
               .css({left: labelx-leftPlus, top: labely, position: 'absolute',
                    padding: 0})
               append(labeltext);
            counter+=fraction;
         }
      } //end of pie
};
```

Adding the Legend

The last part of the code to be defined in your library is that which implements the legend component (see Listing 7-13). Because this part is the same for all three types of charts, it is not subjected to the `if()` statement. Note that within the code, the series are read by the `options.series`, and the colors, by the `options.colors` array.

Listing 7-13. myLibrary.js

```
function myLibrary(target,data,options){
    ...
    if(options.type === 'pie'){
       ...
    } //end of pie
```

```
    //draw the legend
    var legendList = $('<ul class="legend"></ul>')
       .insertBefore(canvas);
    for(var i in options.series){
       $('<li>'+ options.series[i] +'</li>')
       .prepend('<span style="background: '+ options.colors[i] +'" ></span>')
       .appendTo(legendList);
    }
};
```

If you assign the three values `'pie'`, `'line'`, and `'bar'` to the `type` property, you will have the same three charts that you obtained in the previous sections, except that the code that generates them is condensed into one unique version: the `myLibrary.js` library. In addition, you now have the advantage of being able to configure everything from outside the library.

Default Values

But, let us say you forget to define a parameter in `options`. What happens then? The page you are launching would definitely not work correctly. In fact, all libraries, including your very simple one, must contain all the values defined within `options`, which in turn must be already defined with a default value. This is a very important concept, and you will see it with the jqPlot and Highcharts libraries.

You have learned that all graphic elements can be characterized by standard parameters and that these, one by one, create a tree structure property attribute that you can find in the `options` object. Any library, no matter how simple or complex, has such an internal structure. Indeed, each library must provide its `options` structure, in which each property is already specified with a default value, so that if this property is not stated in the `options` object and passed as an argument to the `myLibrary()` function, you do not get any error, because a value would already be assigned to that property. However, the reason for this is not merely to ensure that your library runs if you forget to enter a value, but to get maximum results with minimum effort. Imagine a much more complex library than the one you have just implemented, a library in which the properties to be defined are several. A library of this kind may well be jqPlot. As you will see, you need to define only a few lines to get great results. In fact, it will be enough to write only the parameters you want to change; this spares you a lot of time and effort.

To better understand this concept, if you do not wish to define a value in the property `barGroupMargin`, because, for example, its default value of 4 suits your needs, then you simply do not have to write any reference to it within the `options` object, as shown in Listing 7-14.

Listing 7-14. ch7_01b.html

```
var options = {
    type: 'bar',
    categories: ["May 2012", "Jun 2012", "Jul 2012",
                 "Aug 2012", "Sep 2012", "Oct 2012"],
    series: ["USA", "Canada", "Australia", "Brazil"],
    colors: ['#be1e2d', '#666699', '#92d5ea', '#ee8310'],
    margins: {top: 30, right: 10, bottom: 10, left: 30},
    bar: {}
}
```

And, the library, with appropriate modifications placed for handling the absence of this value, assigns the default value of 4 (see Listing 7-15).

Listing 7-15. myLibrary.js

```
function myLibrary(target,data,options){
    ...
    if(options.type === 'line'){
       ...
    } // end of LINE

    if(options.type === 'bar'){
       if(typeof options.bar.barGroupMargin!= 'undefined') {
          var barGroupMargin = options.bar.barGroupMargin;
       } else {
          var barGroupMargin = 4;
       }
       for(var i in data){
          ctx.beginPath();
          var n = data.length;
       ...
    } // end of bar
    ...
};
```

Figure 7-4 shows the three types of charts that you have implemented in the last three chapters, but this time the code for generating them is all in a single file.

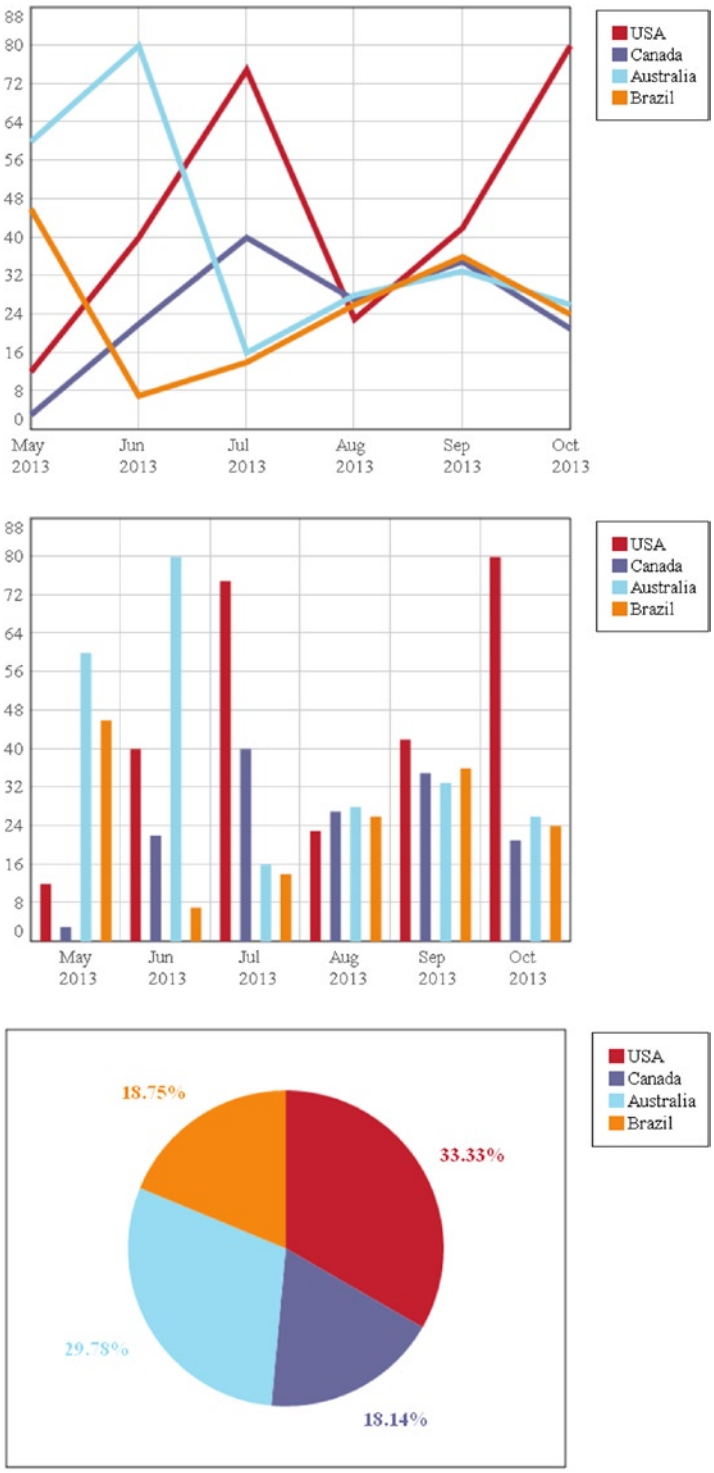


Figure 7-4. *The library generates three type of charts: (a) a line chart, (b) a bar chart, and (c) a pie chart*

Summary

With this chapter, you have concluded the first part of the book. You have seen how it is possible to create a library in JavaScript that is specific to the representation of data in charts.

Learning to reuse an already implemented code to develop a library of your own, you have come to understand how a library of this kind is structured and what the functions are that the various parts perform. Of particular importance is the introduction to the tree structure, which we refer to as `options`, common to many libraries. The `options` object plays an important role in defining all the settings of your graphical components and therefore in defining how your chart will be represented.

In addition, you have seen how data, via an input array, are managed internally for this class of libraries, the role that the jQuery library plays, and the reason why the data are structured on it.

In the next chapter, you will start the second part of the book, in which the **jqPlot** and **Highcharts** libraries will be discussed in full. These libraries are enjoying some success among web developers. Although they are much more complex and feature rich than the library you have just developed, with them you will find all the concepts covered in this chapter.

CHAPTER 8

Introducing jqPlot

With this chapter, you start the second part of the book, concerning the jqPlot library. In the course of this chapter, you will be introduced to the basic concepts that underlie this library. After seeing how the library is structured and the files that compose it, you will begin to understand how easy it is to make a chart using only a few lines of code.

With a series of examples, and through the use of plug-ins, you will gradually learn how to represent any type of chart. Everything will be done using the `$.jqPlot()` function, whose three arguments characterize all the features of the jqPlot library: the target canvas, the input data array, and the options object.

Finally, after a brief illustration of how to customize a chart through the use of Cascading Style Sheets (CSS) styles, you will take a quick look at how thinking in modules can make your implementations ordered, maintainable, and reusable. Let us, therefore, begin our introduction to this wonderful library.

The jqPlot library

jqPlot is a JavaScript library specialized for the generation of charts in web pages. Written completely in pure JavaScript, jqPlot is an open source project, fully developed and maintained by Chris Leonello since 2009. When extended, the jQuery library reaches its full potential functionality. It is for this reason, in addition to its simplicity, that jqPlot is one of the most popular libraries for the representation of charts today.

jqPlot has been very successful and has virtually supplanted other, previous libraries, such as Flot, many aspects of which, including look and feel, jqPlot retains. In fact, the author of jqPlot often admits that he was a dedicated user of Flot but that, over time, he came to realize its limitations. The old library lacked many capabilities; moreover, its architecture was structured in such a way as to make it difficult to expand. So, Leonello felt the need to create a new library that preserved all that was good in Flot but that allowed it to grow. As such, he rewrote its architecture completely. jqPlot has a highly modular structure and, as you will see, is based on a large number of plug-ins, each of which plays a certain role. Hence, its strongest feature is its pluggability. Every object the user draws, be it a line, an axis, a shadow, or the grid itself, is handled by a plug-in. Every plot element has customizable setting options, and every added plug-in can expand the functionality of the plot.

The plug-ins gradually increased in number, widening the library's targets further. jqPlot is now a versatile and expandable library, suitable for those who want to develop professional charts in a just a few steps.

In most cases, jqPlot allows you to draw beautiful charts without adding too many lines of code. Indeed, you will see that jqPlot (perhaps even more than jQuery) has embraced the philosophy "Write less, do more." I think that this is the libary's most appreciated aspect. Every day, more and more developers are added to the list of jqPlot users.

Including Basic Files

When you decide to take advantage of jqPlot to draw a chart on your web site, there is, as a starting point, a set of critical files that needs to be included.

As mentioned earlier, jqPlot is essentially an extention of jQuery, and so operating with it requires the inclusion of the jQuery plug-in (see Table 8-1). You can download this plug-in from the official jqPlot web site (www.jqplot.com), along with all the other plug-ins that make up the jqPlot library, including the CSS file. These files are grouped in different distributions, depending on the release version.

Table 8-1. *The distributions of jqPlot and versions of jQuery on which they are based*

jqPlot Version	jQuery Version
1.0.6–1.0.8	1.9.1
1.0.2–1.0.5	1.6.4

■ **Note** All the examples in this book use version 1.0.8 of the jqPlot library.

However, there is a small set of files that represents the core of the library and that is indispensable if you want to include every function made possible by jqPlot. This set of basic files consists of the jQuery plug-in, the jqPlot plug-in, and a jqPlot CSS file. There is another file that needs to be imported, but only if you want to load the page in an Internet Explorer browser that is below version 9: an ExplorerCanvas (excanvas) script. This optional file compensates for the lack of canvas functionality introduced by HTML5.

Thus, within the <head></head> tags of your web page, you are going to include these files (for further information on how to set up a workspace, see Appendix A):

```
<!--[if lt IE 9]><script type="text/javascript" src="../src/excanvas.js"></script><![endif]-->
<script type="text/javascript" src="../src/jquery.min.js"></script>
<script type="text/javascript" src="../src/jquery.jqplot.min.js"></script>
<link rel="stylesheet" type="text/css" href="../src/jquery.jqplot.min.css" />
```

Instead of working with the jqPlot library locally, by downloading it from the web site, you can also use a content delivery network (CDN) service, just as you have done with jQuery. jsDelivr (www.jsdelivr.com/#!jqplot) is a CDN web site that offers all the most recent distributions of jqPlot.If you want to use this service, you can modify the URL as follows:

```
<!--[if lt IE 9]><script type="text/javascript"
src="http://cdn.jsdelivr.net/excanvas/r3/excanvas.js"></script><![endif]-->
<script src="http://code.jquery.com/jquery-1.9.1.min.js"></script>
<script type="text/javascript"
src="http://cdn.jsdelivr.net/jqplot/1.0.8/jquery.jqplot.min.js"></script>
<link rel="stylesheet" type="text/css"
href="http://cdn.jsdelivr.net/jqplot/1.0.8/jquery.jqplot.min.css" />
```

As you will soon discover, in the network, you will come across similar files, some with and others without the abbreviation “min” (for “minified”). You should always try to use the min versions, the compressed versions of these files. They are faster to load. You should use their normal versions (without “min”) only when your intent is to modify these libraries internally.

Plot Basics

Now that you have seen how the jqPlot library is structured and have looked at the set of files you require to operate with it, you can begin to learn how to use these files within your web page. In chart development, there are two basic steps: the creation of an area in which to represent the chart and the insertion of a section for the JavaScript code needed to call all the functions, variables, and series of objects that the jqPlot library makes available.

Adding a Plot Container

In Chapters 3–7, you saw how the canvas was used as a drawing area in which to develop your charts. Similarly, the jqPlot library requires the definition of a container (a element) within the `<body>` section of the HTML page. This container will function as a canvas for the library.

Throughout this chapter, this container is referred to as the **target**. Within a web page, each target (in this case, the plot container) is identified by a specific `id`. In this book, you will always find `myChart` as an identifier of the target (, but it can take any name, and you must always bear in mind that more than one target may be assigned to the same web page. Furthermore, it is also important to specify a width and a height for the target. These will define the size of the drawing area within the web page (see Listing 8-1).

Listing 8-1. ch8_01.html

```
<BODY>
   ...
   <div id="myChart" style="height:400px; width:500px;"></div>
   ...
</BODY>
```

Creating the Plot

To be entered, jqPlot commands and almost all JavaScript code must be enclosed within a `<scripts>` tag. But, a web page is divided into two sections: head and body. So, which is the best place to insert JavaScript code? Although it is possible to place code in both sections of a library, it is preferable to place it in only one, depending on the library. Considering how jqPlot works, you will be putting the code between the `<head></head>` tags.

Furthermore, jqPlot is an extension of jQuery, so you need to call all its methods inside the `$(document).ready()` function if you want your code to be executed (see Listing 8-2).

Listing 8-2. ch8_01.html

```
$(document).ready(function(){
    // Insert all jqPlot code here.
});
```

Then, to create the actual plot, you must call the `$.jqplot` plug-in with the `id` of the target in which you want to draw the chart. This call is executed by the following jQuery function:

```
$.jqplot(target, data, options);
```

The `jqplot()` function has three arguments; **target**, which is the ID of the target element in which the plot is to be rendered—the ID attribute that you specify in the plot container; **data,** consisting of an array for data series; and **options**, the main feature of jqPlot. Within options, you will enter the customization settings necessary to make your chart more suitable to your needs and tastes.

If you do not define any options (yes, it is optional!), you can produce a chart following the settings for standard options. In fact, many options are already defined, and it is not necessary to change the all settings every time you develop a chart. If the standard options meet your requirements, you do not need to define them. This will save you a lot of time and avoids having to write many lines of code. For example, let us write the function in Listing 8-3.

Listing 8-3. ch8_01.html

```
$(document).ready(function(){
    $.jqplot ('myChart', [[100, 110, 140, 130, 80, 75, 120, 130, 100]]);
});
```

With only a few lines of code, you can produce charts like the one in Figure 8-1.

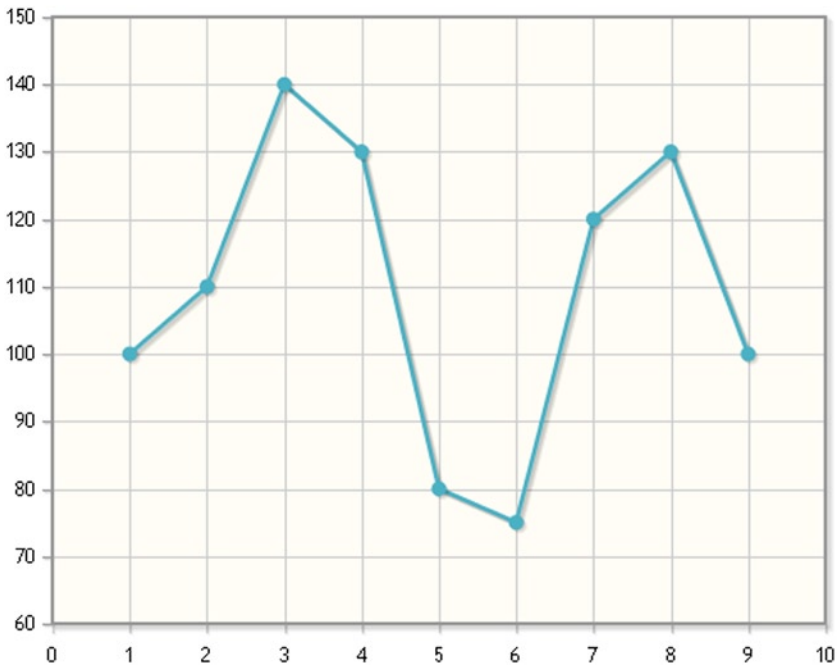


Figure 8-1. *A line chart created with only a few lines of code*

From what you have just seen, you can surmise that if you do not specify any options, the default outcome will be a line chart, and the data that you have added will be interpreted as such. The values in the array are, therefore, the y values, and the indexes of their sequence are reported on the x axis. In later chapters, you will learn how these values are interpreted and how to get different types of charts from the linear one.

Using jqPlot Plug-ins

The most recent jqPlot distribution offers approximately thirty plug-ins (for a list of all the jqPlot plug-ins, see Appendix B). Each is specialized to perform a specific task, and the name is often indicative of function. You will be looking at many of these plug-ins in the following chapters—at their uses and their main options.

Let us take, for instance, *BarRenderer*. This plug-in is necessary if you want the input data to be interpreted as a bar chart:

```
$.jqplot ('myChart', [[100, 110, 140, 130, 80, 75, 120, 130, 100]],
{
    series:[{renderer: $.jqplot.BarRenderer}]
});
```

In jqPlot, we often refer to a plug-in as a renderer. This is because the architecture of the framework specifies that each plug-in must cover a specific task. If the developer deems it necessary, then he or she will include it. In addition, real renderer should be as independent as possible from one another. In fact, you can add as many plug-ins as you wish, and generally their order is not important. Some plug-ins do not require that you specify any extra option or setup; they are already defined and are directly activated just by virtue of being included. One such plug-in is *Highlighter*, which highlights data points near the mouse. However, if you are not satisfied with the default settings, you can always define the properties with new values; these plug-ins also contain additional, settable properties. Other plug-ins provide functionalities that have to be specified in the options argument in order to be activated.

Thus, both the basic elements of the jqPlot library and the additional components, which are introduced gradually by the included plug-ins, can be characterized by a series of attributes (in a manner very similar to that seen with the CSS style). The jqPlot library calls these attributes **options**.

Understanding jqPlot Options

The key to using jqPlot effectively is to understand jqPlot's options. The properties of any object in a chart are defined by attributes, which can take different values. It is very important to understand how to set and use these attributes through object types that I will refer to as **options**.

Inserting Options

So far, you have seen how the `jqPlot()` function is called within the JavaScript code and how to include the plug-ins and data, but you have not yet observed how to enter options. You can customize the default line chart by passing different attributes to the `$.jqplot()` function in this way:

```
$(document).ready(function(){
    $.jqplot ('myChart', [[100, 110, 140, 130, 80, 75, 120, 130, 100]],
    {
        //All the attributes here.
    });
});
```

The first thing to note is that it is not possible to set properties directly in the chart object after you call `$.jqplot()`. At best, this will not do anything. You have to pass all the attributes in the options argument.

The options argument represents the `jqPlot` object in each of its properties. Everything that characterizes a chart is expressed by a number of properties, which are set to certain values. These values differentiate a bar chart from a line chart, regulate the stroke of a line or the length of an axis, indicate whether to display a legend and where, and so on. Usually, when including the various plug-ins, it is not necessary to specify values for all properties; they are already set to a default value. It is because of these default values that, in adding a plug-in, you can realize a nice chart without adding a single line of code.If you specify a property explicitly, you are actually overwriting the value of a property already defined with a default value.

Because our aim is to set the `jqPlot` object, and because this is made up of a series of components, it will be necessary to build the `options` object with a structure that perfectly reflects these components by defining a whole series of objects with their properties. Recalling the object corresponding to the component and assigning a value to one of its properties inside the `options` object, you are going to overlay the default value and change the property of the respective component of the `jqPlot` object. The most commonly used objects you can define inside the `options` object are

- `seriesColors`
- `stackSeries`
- `title`
- `axesDefaults`
- `axes`
- `seriesDefaults`
- `series`
- `legend`
- `grid`
- `cursor`
- `highlighter`

Each name reflects the component of the chart that will be affected by a change in its property value. These objects are built by a whole series of well-defined properties, each with its own default value.

This is the structure of the `jqPlot` object:

jqplot object > component objects > object properties > default value

In Listing 8-4, you can see the corresponding structure that you need to follow when defining the options object.

Listing 8-4. ch8_02c.html

```
var options = {
    axes:{
       yaxis:{
          min: 70,
          max: 150
       },
       ...
    },
    ...
};
```

I believe that the easiest object to add to a chart is the `title`. It does not contain any property, and can often be considered a property itself of the `jqPlot` object. Moreover, it is possible to set a text value directly on it, text that will be the title of your chart. Given its simplicity, the `title` is a good starting point for understanding how to use options (Listing 8-5).

Listing 8-5. ch8_02a.html

```
$(document).ready(function(){
    $.jqplot ('myChart', [[100, 110, 140, 130, 80, 75, 120, 130, 100]],
    {
        title: 'My first jqPlot chart'
    });
});
```

If you prefer, you can also define the properties of objects externally, with the `jqplot()` function, assigning properties to a variable. This variable will then be passed as an argument in the `jqPlot()` function, as shown in Listing 8-6. This variable is actually the `options` object.

Listing 8-6. ch8_02a.html

```
var options = { title: 'My first jqPlot chart' };
$.jqplot ('myChart', [[100, 110, 140, 130, 80, 75, 120, 130, 100]], options);
```

In both cases, you now have a chart with a title at the top (see Figure 8-2).

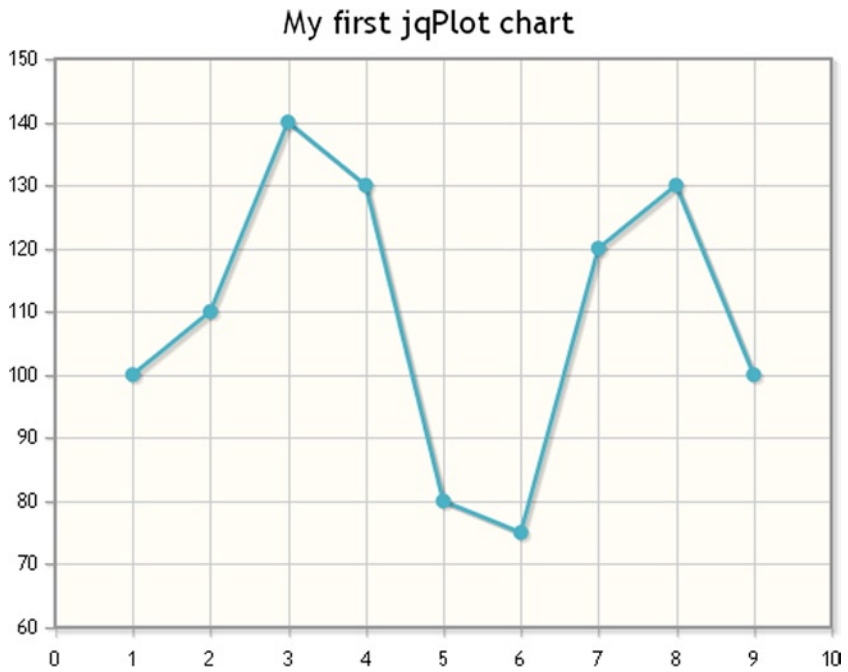


Figure 8-2. *Adding a title to a line chart*

To better understand how to set the jqPlot properties within the `options` object, let us take an example, referring to the API Documentation section of the jqPlot web site (`www.jqplot.com/docs/files/jqplot-core-js.html`). Let us say you want to hide the grid lines in your chart. In the list of properties belonging to the `grid` object, you will find what you are looking for:

```
this.drawGridlines = true.
```

`this` is the instance of the grid, and `true` is the default value assigned to the `jqplot` object at the time of its creation. Because you want the grid to be hidden (a behavior different from that of default), you will need to replace the value `true` with the value `false` within the `jqPlot` object. To do this, you have to add the `drawGridlines` property within the `options` object definition, maintaining the structure object:{property:attribute}.

```
options = {grid:{drawGridlines: false}};
```

Now, you have a chart without grid lines (see Figure 8-3).

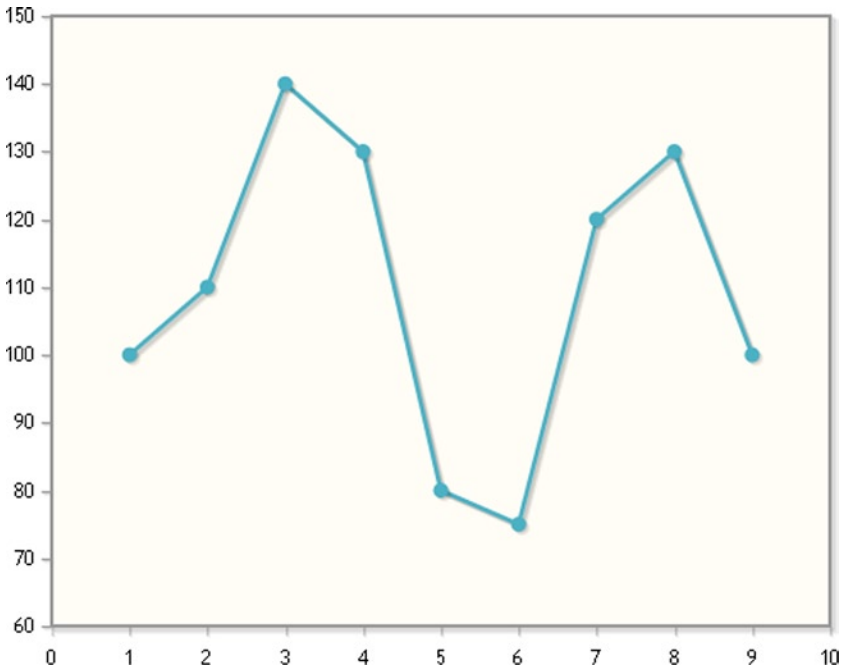


Figure 8-3. *Hiding the grid lines in a line chart*

For a full list of attributes that can be set, you can go to the official jqPlot web site (`www.jqplot.com/docs/index/General.html`) or read the `jqPlotOptions.txt` file contained in each distribution.

Handling Options on Axes

Axes are handled a little differently from the other normal component objects, because they have four distinct children, namely, `xaxis`, `yaxis`, `x2axis`, and `y2axis`. To illustrate axes, we therefore need a more deeply nested example. Let us say you want to specify the `min` and `max` attributes on the y axis. To do so, you will need to specify the `options` object with the structure shown in Listing 8-7.

Listing 8-7. ch8_02c.html

```
var options = {
    axes:{
        yaxis:{
            min:70,
            max:150
        }
    }
};
```

Now, the range on the y axis is between the `max` and `min` properties that you defined in `options` (see Figure 8-4).

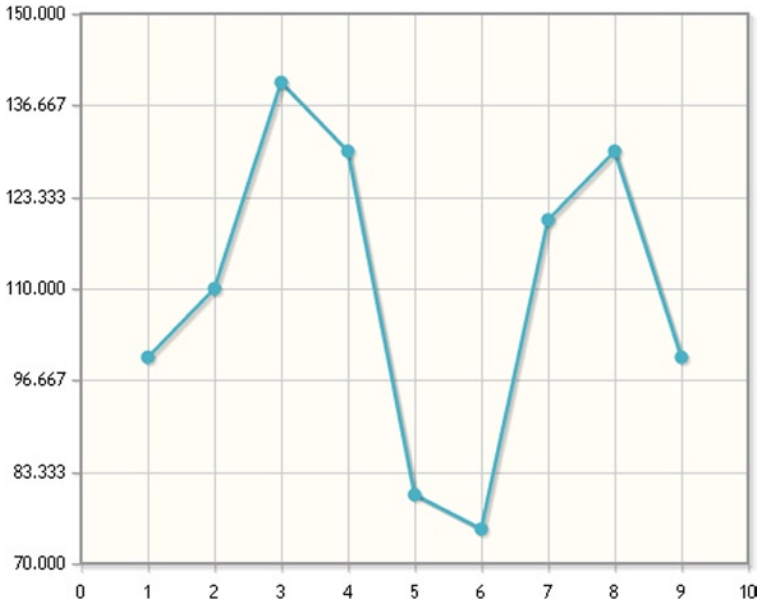


Figure 8-4. *A line chart focused on a specific range on the y axis*

To make things easier, jqPlot provides a handy shortcut that enables us to assign the same value to the properties of all axes in one go: the `axesDefaults` object. If you want to set the same value to both x and y (or x2 and y2), you need to specify these properties only for the `axesDefaults` option object, assigning the value once (see Listing 8-8).

Listing 8-8. ch8_02d.html

```
$(document).ready(function(){

   var options = {
       axesDefaults:{
          min:0,
          max:20
       }
   };
   $.jqplot ('myChart', [[1,4,8,13,8,7,12,10,5]], options);
});
```

Inserting Series of Data

Earlier, you saw how to produce a simple line chart, using the settings for standard options (see the section "Creating the Plot"). In this example the array of data was passed directly as the second argument in the function `$.jqplot()`. However, you can also define an array of data as a variable externally and then pass it as the second argument.

```
$(document).ready(function(){
    var data = [[100,110,140,130,80,75,120,130,100]];
    $.jqplot ('myChart', data);
});
```

Here, you find a single series of data corresponding to the values on the y axis. But, as you will see, it is possible to pass data as (x, y) pairs of values and also to pass multiple series of data at once. These input modes vary, depending on the chart you are designing and the requirements of the various plug-ins used. For example, if you want to input multiple data series, you need to declare four different arrays, as shown in Listing 8-9.

Listing 8-9. ch8_03a.html

```
$(document).ready(function(){
    var series1 = [1, 2, 3, 2, 3, 4];
    var series2 = [3, 4, 5, 6, 5, 7];
    var series3 = [5, 6, 8, 9, 7, 9];
    var series4 = [7, 8, 9, 11, 10, 11];
    $.jqplot ('myChart', [series1, series2, series3, series4]);
});
```

jqPlot is able to manage multiple series without having to specify any property in `options`. In fact, the browser will display a line chart with as many lines as there are data series, each in a different color, as shown in Figure 8-5.

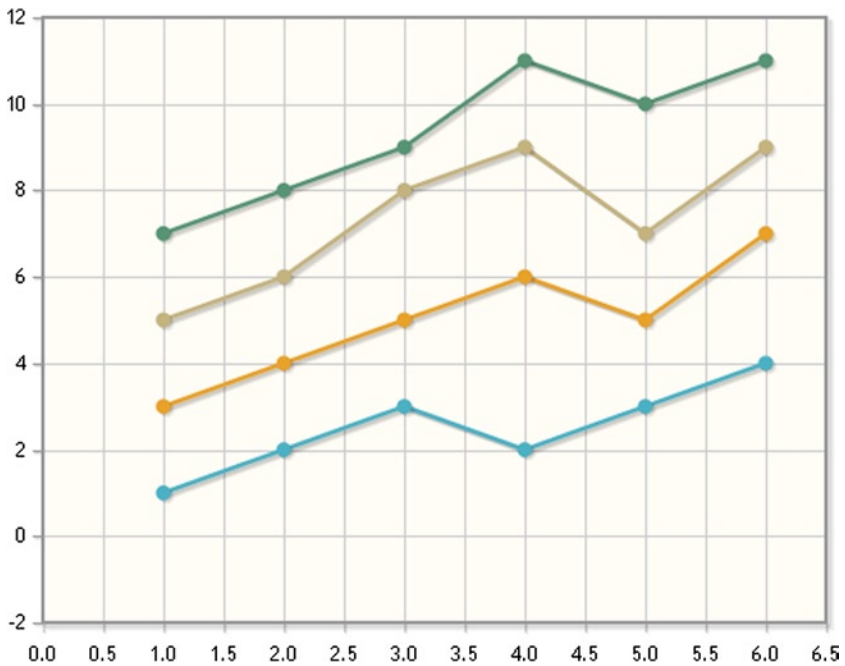


Figure 8-5. *A multiseries line chart with different colors for each series*

As you can see, as well as making the code more readable and tidy, the externally defined series of data can also assist in the future extension and manipulation of the data. Using all the tools that JavaScript makes available, you can create, manipulate, sort, calculate, and compute an infinite variety of data.

It is possible to change the properties of the `options` object even for series of data. The series are inserted in a particular order into an array passed as a second argument in the `$.jqplot()` function. This order will be reflected in the creation of the `series` objects inside the `jqPlot` object. For instance, if you want only the second series not to show its marker points, it will be necessary to leave empty the space for the properties of the first series (not to overwrite its attributes) and then set the `showMarker` property to `'false'` in the second space. In so doing, jqPlot will overwrite only the values of the property of the second series. To accomplish this, you must write the `options` object as shown in Listing 8-10.

Listing 8-10. ch8_03b.html

```
$(document).ready(function(){
    var series1 = [1, 2, 3, 2, 3, 4];
    var series2 = [3, 4, 5, 6, 5, 7];
    var series3 = [5, 6, 8, 9, 7, 9];
    var series4 = [7, 8, 9, 11, 10, 11];
    var options = {
       series: [ {},
              {
                showMarker: false
              }]
       }

      $.jqplot ('myChart', [series1, series2, series3, series4], options);
});
```

The result of these settings is the chart with four series in Figure 8-6. Note that the third series from the top has no marker point.

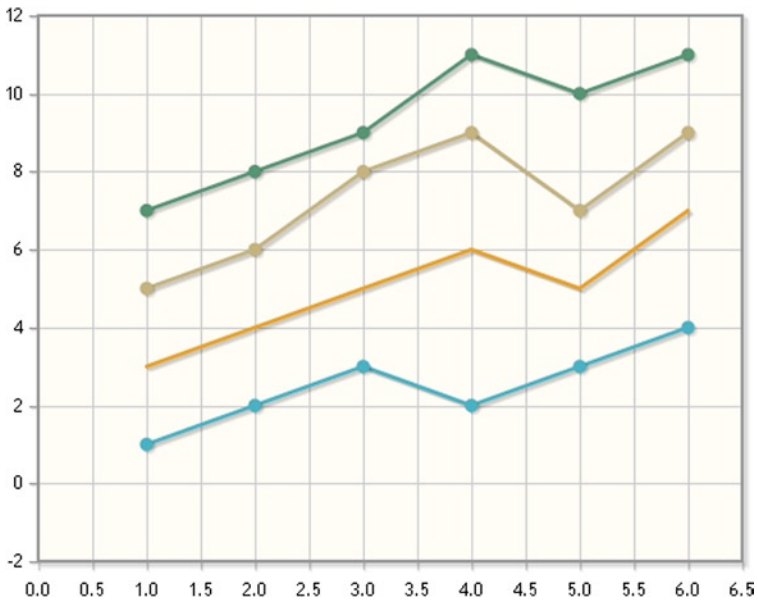


Figure 8-6. *A multiseries chart with a series showing no markers*

If you decide to set the `showMarker` property in `axesDefaults`, instead of in the `axes` object, you will assign the same value for all the series at once (see Listing 8-11).

Listing 8-11. ch8_03c.html

```
$(document).ready(function(){
    var series1 = [1, 2, 3, 2, 3, 4];
    var series2 = [3, 4, 5, 6, 5, 7];
    var series3 = [5, 6, 8, 9, 7, 9];
    var series4 = [7, 8, 9, 11, 10, 11];
    var options = {
       seriesDefaults: { showMarker: false }
    };
    $.jqplot ('myChart', [series1, series2, series3, series4], options);
});
```

Now, none of the series in the chart show any marker points (see Figure 8-7).

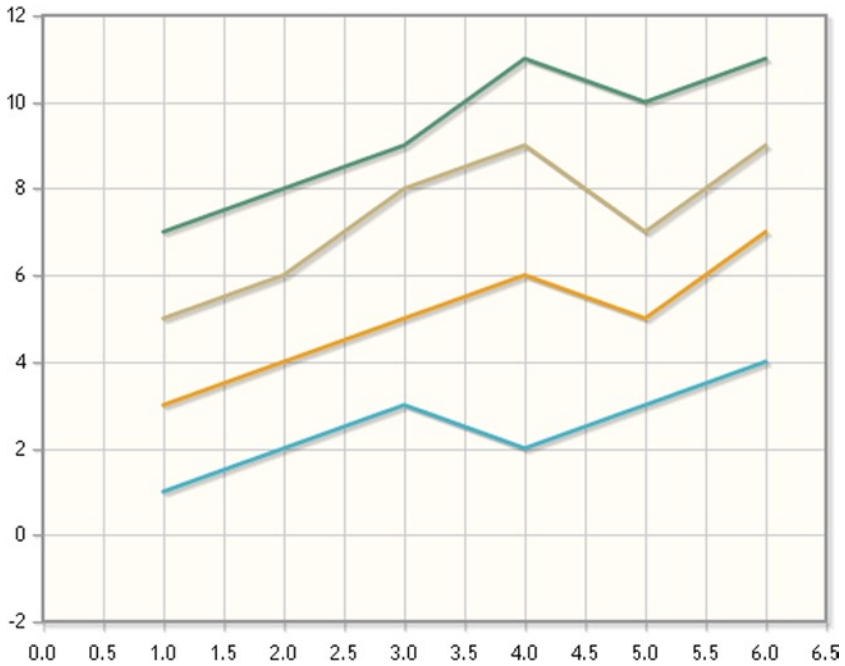


Figure 8-7. *A multiseries chart with no markers*

There is a third way to enter the data as an array. You have just seen a case in which an array is defined for each series and then passed to the `$.jqplot()` function, all gathered in one array:

```
var series1 = [1, 2, 3, 2, 3, 4];
var series2 = [3, 4, 5, 6, 5, 7];
var series3 = [5, 6, 8, 9, 7, 9];
var series4 = [7, 8, 9, 11, 10, 11];
$.jqplot ('myChart', [series1, series2, series3, series4], options);
```

But, it is also possible to define all the series in a single variable, which you call `dataSets`:

```
var dataSets = {
   data1: [[1,1], [2,2], [3,3], [4,2], [5,3], [6,4]],
   data2: [[1,3], [2,4], [3,5], [4,6], [5,5], [6,7]],
   data3: [[1,5], [2,6], [3,8], [4,9], [5,7], [6,9]],
   data4: [[1,7], [2,8], [3,9], [4,11], [5,10], [6,11]]
};
```

Once you have declared the `dataSets` variable, in order to access the values, you have to specify the series inside it, with `dataSets.` as a prefix. Thus, when you need to pass the four series individually as a second argument of the `jqplot()` function, you must do so in this way:

```
$.jqplot ('myChart', [dataSets.data1, dataSets.data2, dataSets.data3, dataSets.data4], options);
```

Although, at the moment, this whole operation may seem too laborious, later you will see that gathering all the data in a data set can be useful in special cases.

Renderers and Plug-ins: A Further Clarification

Normally, a renderer is an object that is attached to something in the plot in order to draw it. A plug-in, as well as adding drawing functionality, can perform other functions, such as event handling; making calculations; and handling the format of strings and values, such as dates. So, it is possible to consider a renderer a drawing plug-in, but the converse is not always true.

Let us examine this slight difference in more detail with the help of some examples. You have seen, for instance, that by entering only a single data series, you can obtain a line chart by default (see the section "Creating the Plot"). If you want to render this series as a bar chart, you need to attach the *barRenderer* plugin to the `seriesDefaults` object in `options`. Moreover, when switching from a line chart to a bar chart, it is necessary to create categories on the x axis in order to have the bars well separated from each other. To do this, you need to attach the `CategoryAxisRenderer` to the `axes` object in `options` (see Listing 8-12).

Listing 8-12. ch8_04a.html

```
$(document).ready(function(){
    var data = [[100, 110, 140, 130, 80, 75, 120, 130, 100]];
    var options = {
       seriesDefaults: {
         renderer: $.jqplot.BarRenderer
       },
       axes:{
         xaxis:{
           renderer: $.jqplot.CategoryAxisRenderer
         }
      }
    }
    $.jqplot ('myChart', data, options);
});
```

However, calling the two renderers in `options` is not enough. You must also load them in the page, so you have to include the corresponding plug-ins, as shown in Listings 8-13 and 8-14 (CDN service).

Listing 8-13. ch8_04a.html

```
<script type="text/javascript" src="../src/jquery.min.js"></script>
<script  type="text/javascript" src="../src/jquery.jqplot.min.js"></script>
<link rel="stylesheet" type="text/css" href="../src/jquery.jqplot.min.css" />
<script type="text/javascript" src="../src/plugins/jqplot.barRenderer.min.js"></script>
<script type="text/javascript" src="../src/plugins/jqplot.categoryAxisRenderer.min.js"></script>
```

Listing 8-14. ch8_04a.html

```
<script src="http://code.jquery.com/jquery-1.9.1.min.js"></script>
<script type="text/javascript"
src="http://cdn.jsdelivr.net/jqplot/1.0.8/jquery.jqplot.min.js"></script>
<link rel="stylesheet" type="text/css"
href="http://cdn.jsdelivr.net/jqplot/1.0.8/jquery.jqplot.min.css" />
<script type="text/javascript"
src="http://cdn.jsdelivr.net/jqplot/1.0.8/plugins/jqplot.barRenderer.min.js"></script>
<script type="text/javascript"
src="http://cdn.jsdelivr.net/jqplot/1.0.8/plugins/jqplot.categoryAxisRenderer.min.js"></script>
```

If you reload your page in the browser, the line chart has just become a bar chart, as shown in Figure 8-8.

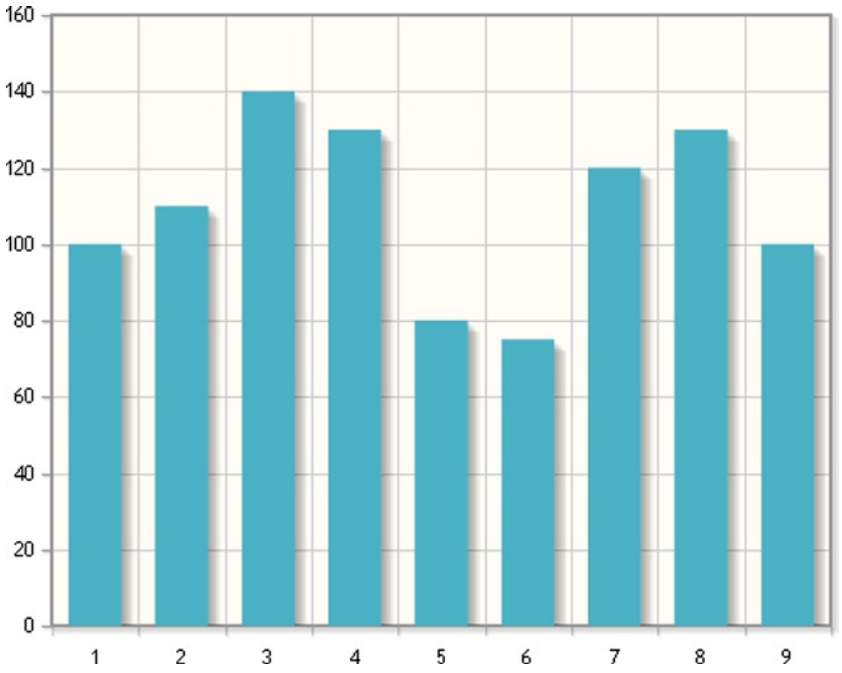


Figure 8-8. *A bar chart*

By calling these two renderers to `options`, you replace the default renderer valid for all series in the plot with these category renderers. The latter have in turn several properties set on default values, which may be modified as well. Many of these properties may be specific to that particular renderer, and so they will be added to those already defined in the `jqplot` object in order to introduce new features to the plot.

Even for this class of additional properties, you can change the default values through the `options` object. First, assign the desired renderer to the `renderer` property. Then, specify all the properties you want to set inside the `rendererOptions` property. All these properties will be specified in the component object on which you want to act. For example, if you want each bar of a given series to have a different color, you need to change the `varyBarColor` property, replacing the default value `false` with `true` (see Listing 8-15).

Listing 8-15. ch8_04b.html

```
var options = {
    seriesDefaults: {
        renderer: $.jqplot.BarRenderer,
        rendererOptions: {
           varyBarColor: true
        }
    },
    axes:{
       xaxis:{
          renderer: $.jqplot.CategoryAxisRenderer
      }
    }
}
```

With the changes you have just made, the *BarRenderer* plugin will automatically assign a different color to each bar (see Figure 8-9).

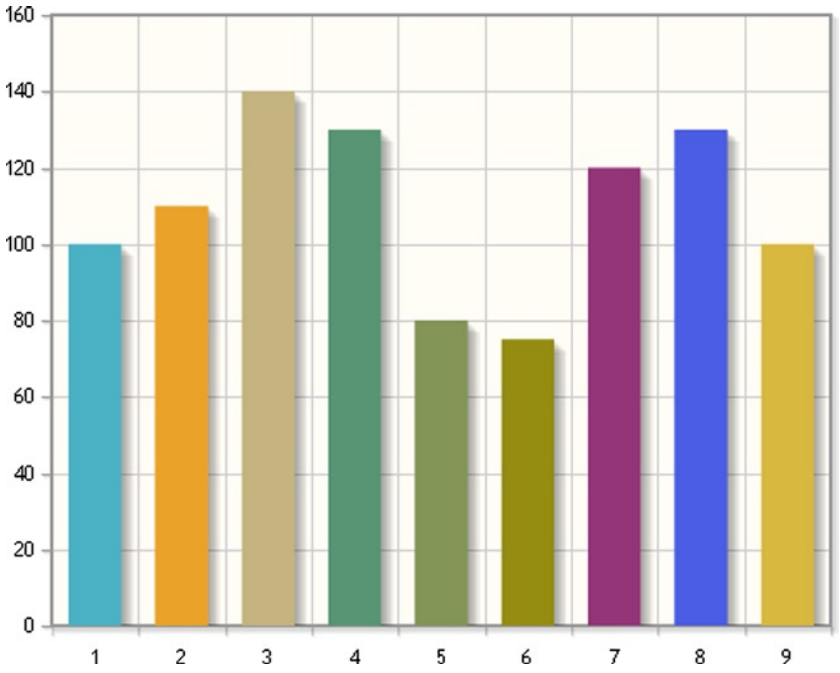


Figure 8-9. *A bar chart with different colors*

Plug-ins also have specific properties that can be set inside the `options` object. As mentioned earlier, not all jqPlot plug-ins are renderers, and those that are not are easily recognizable in the jqPlot distribution, because they do not contain the term *renderer* in their file names. These plug-ins perform specific functions that are not directly related to a particular type of component in the plot. Such features enhance the capability of jqPlot in general. *Highlighter*, for instance, is a plug-in that highlights data points when they are moused over. As you will see, this plug-in has a series of tools within it that handles formatting specifiers for data values and that can show tool tip content with an HTML structure. Other notable plug-ins include *Trendline*, which automatically computes and draws trend lines for plotted data; *Cursor*, which represents the cursor, as displayed in the plot; and *PointLabels*, which places labels at the data points.

CSS Customization

Much of the styling of jqPlot charts is done through CSS. The `jquery.jqplot.css` file is available in every distribution, and it is one of the three fundamental files to be included in your web page in order to obtain a jqPlot chart.

All the components that make up the chart can be customized through CSS without having to set any of their properties in the `options` object. This is to maintain consistency with all other objects in the web page: the style of the chart, and all that is in it (inside the canvas), must be managed by CSS files, like any other HTML object. The names of the CSS classes ruling the style of jqPlot objects begin with the prefix `.jqplot-*`. For example, the style class that affects all axes is `.jqplot-axis`.

To illustrate how it is possible to modify some elements of the chart using CSS, let us look at how to change the font and font size of the chart title. As with any HTML element, you simply have to recall the CSS selector of the `jqPlot` element and modify the attributes. So, in this case, you add the CSS style setting in Listing 8-16.

Listing 8-16. ch8_02e.html

```
<style>
    .jqplot-title {
        font:italic bold 22px arial,sans-serif;
    }
</style>
```

With this new CSS statement, you have changed the style of the title, as Figure 8-10 shows.

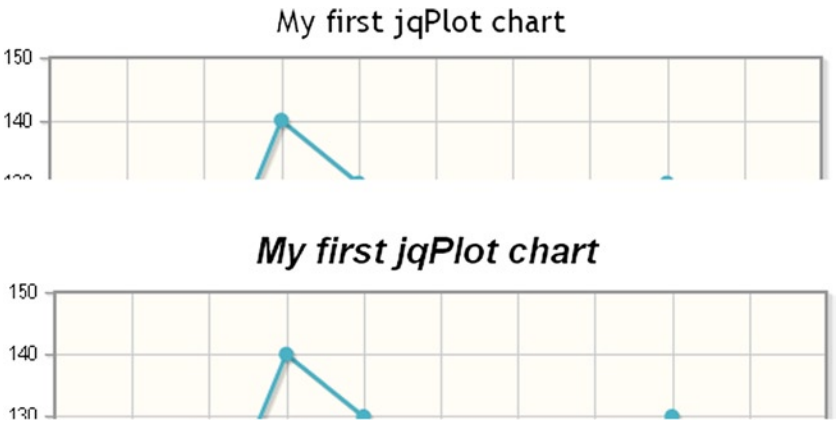


Figure 8-10. *Two different CSS styles applied to the title*

Thinking in Modules

When things get increasingly complex, and the lines to add to your web site become many, it is best to think in terms of modules. As well as providing better visibility and ease of maintenance, creating separate modules also promotes the reusability of what you have just created. Let us analyze the current situation with your web page in Listing 8-17.

Listing 8-17. ch8_05a.html

```
<HTML>
<HEAD>
<TITLE>My first chart</TITLE>
<script type="text/javascript" src="../src/jquery.min.js"></script>
<script  type="text/javascript" src="../src/jquery.jqplot.min.js"></script>
<link rel="stylesheet" type="text/css" href="../src/jquery.jqplot.min.css" />
<style>
    .jqplot-title {
        font: italic bold 22px arial,sans-serif;
    }
</style>
<script class="code" type="text/javascript">
$(document).ready(function(){
    var data = [[100, 110, 140, 130, 80, 75, 120, 130, 100]];
    $(document).ready(function(){
        $.jqplot ('myChart', data,
        {
            title: 'My first jqPlot chart'
        });
    });
});
</script>
</HEAD>
<BODY>
    <div id="myChart" style="height:400px; width:500px;"></div>
</BODY>
</HTML>
```

If you load this web page in the browser, you obtain the line chart in Figure 8-11.

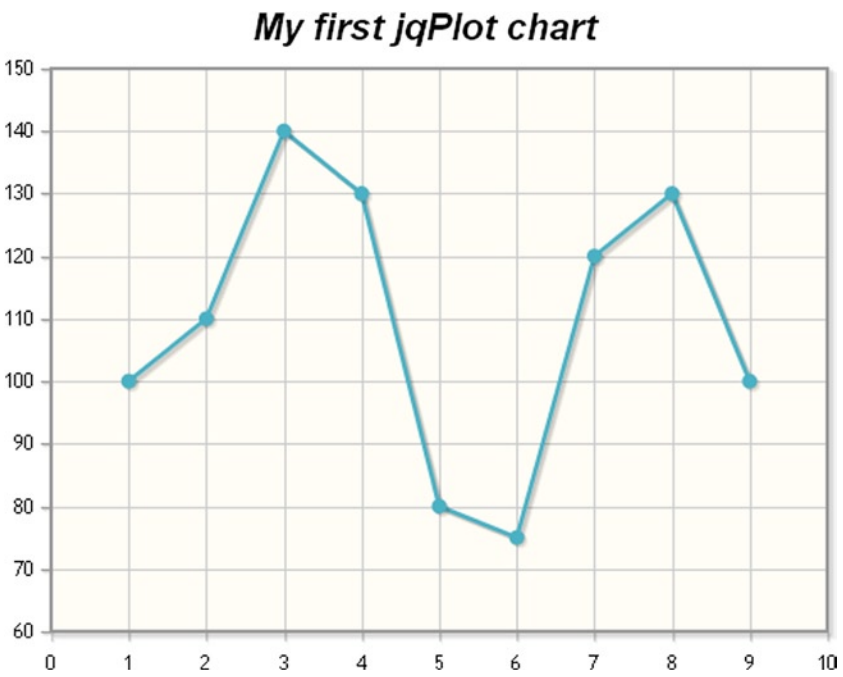


Figure 8-11. *A simple line chart*

As is evident, the part that regulates the style of the page is contained within the <style></style> pair of tags, whereas JavaScript, or the jqPlot code, is within the <script></script> pair of tags. These are the two main areas in which we add lines of code. But, it is interesting to note that these two sections can be copied and pasted into two separate external files. The style section will be copied in a file that we will call `myCss.css` (see Listing 8-18).

Listing 8-18. myCss.css

```
.jqplot-title {
    font: italic bold 22px arial,sans-serif;
}
```

The JavaScript code will be copied in a file that we will name `myJS.js` (see Listing 8-19).

Listing 8-19. myJS.js

```
$(document).ready(function(){
    var data = [[100, 110, 140, 130, 80, 75, 120, 130, 100]];
    $.jqplot ('myChart', data,
    {
       title: 'My first jqPlot chart'
    });
});
```

Now, you can change your HTML page by removing the copied parts and including the two newly created files (see Listing 8-20).

Listing 8-20. ch8_05b.html

```
<HTML>
<HEAD>
    <TITLE>My first chart</TITLE>
    <script type="text/javascript" src="../src/jquery.min.js"></script>
    <script  type="text/javascript" src="../src/jquery.jqplot.min.js"></script>
    <link rel="stylesheet" type="text/css" href="../src/jquery.jqplot.min.css" />
    <script  type="text/javascript" src="myJS.js"></script>
    <link rel="stylesheet" type="text/css" href="myCss.css" />
</HEAD>
<BODY>
   <div id="myChart" style="height:400px; width:500px;"></div>
</BODY>
</HTML>
```

If you now load the page in your browser, you will not see any difference, compared with the initial case, in which the JavaScript code and CSS styles were on the same page (see Figure 8-11).

Being aware of the possibility of working in modules will enable you to write a code that can be reused by multiple charts. This can be very useful when you want to create, for example, a standard set of CSS styles that assigns a graphic theme that will be common to all your charts. Or, if you have developed methods in JavaScript the application of which can be valuable in many other cases, then you can include these methods externally in each HTML page of your personal library.

Summary

In this chapter, you took your first steps with the jqPlot library. In particular, you looked at how the library is designed and at key concepts, such as plug-ins and options. With a series of examples, you gradually learned how to use the three different arguments passed to the function `$.jqPlot()`: the **target**, which is the jqPlot library canvas; input **data**—their format and the various input modes; and, especially, the `options` object, through which the settings of all the components of the library are performed. `options` will constitute the core of all implementations that you will be looking at in later chapters.

In the following chapters, you will be using the jqPlot library more specifically, implementing all the most common chart types. In the next chapter, you will begin with **line charts**.

CHAPTER 9

Line Charts with jqPlot

In the previous chapter, you observed the most basic use of jqPlot, in which a series of data serves to plot a line, with no need for any additional options. You saw that in order to create the most basic type of chart, a line chart, , you do not need to include plug-ins.

In this chapter, you will begin to examine in more detail the possibilities that the jqPlot library affords by exploring the various plug-ins and their functionality. First, because the line chart is represented on the Cartesian axes, you will be introduced to the use of pairs of values (x, y) as input data. You will then move on to the study the axes and how to create them, using the appropriate plug-ins. You will also analyze in detail how to implement the various elements connected to the axes as ticks, axis labels, and grid. A discussion of logarithmic scale (log scale) follows.

Next, you will learn how to realize multiseries line charts through the treatment of multiple series of data at the same time. You will discover how, by setting the lines and markers, you can modify patterns, shapes, and even colors. In addition, you will view how to create an animation by adjusting the speed at which the browser draws the chart.

Moreover, you will investigate the way in which the jqPlot library lets you manipulate different formats of date and time values. You will also see how it is possible to customize some elements, using the HTML format, along with the highlighting of data points. In the final part of the chapter, you will deal with more complex cases, such as generating a trend line and working with band charts.

Using (x, y) Pairs as Input Data

So far, for simplicity's sake, the input data have been passed in as an array of y values (see Listing 9-1). If jqPlot finds y values only, x values are assigned as 1, 2, 3, and so on, following their order in the array.

Listing 9-1. ch9_01.html

```
$(document).ready(function(){
    var plot1 = $.jqplot ('myChart', [[100,110,140,130,80,75,120,130,100]]);
});
```

In Figure 9-1, you can see along the x axis a sequence of integer numbers, which are the indexes of the array passed as data.

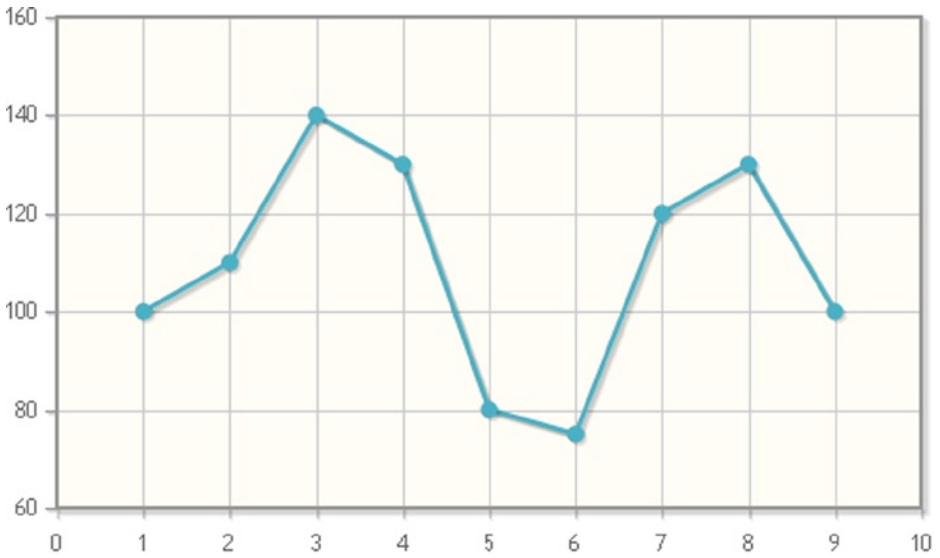


Figure 9-1. *The x axis reports the indexes of the values inserted*

When you are working with a linear plot, it is better to use arrays with pairs of values (x, y), as this avoids many complications, such as the need to enter the data in a particular order, which is not always possible or correct. In fact, using pairs of values, the data should not be listed in order of increasing x value; jqPlot will do that for you. Furthermore, the values of x need not be equidistant, but can follow any distribution. In Listing 9-2, pairs of values (x, y) have been inserted in which the x values are neither sorted nor evenly distributed.

Listing 9-2. ch9_02.html

```
$(document).ready(function(){
    var data = [[[10,100], [80,130], [65,75], [40,130],
                 [60,80], [30,140], [70,120], [20,110], [95,100]]];
    $.jqplot ('myChart', data);
});
```

In Figure 9-2, you can see how jqPlot sorts all the points in the chart, regardless of the order in which they were entered and whether they have been uniformly distributed along the x axis.

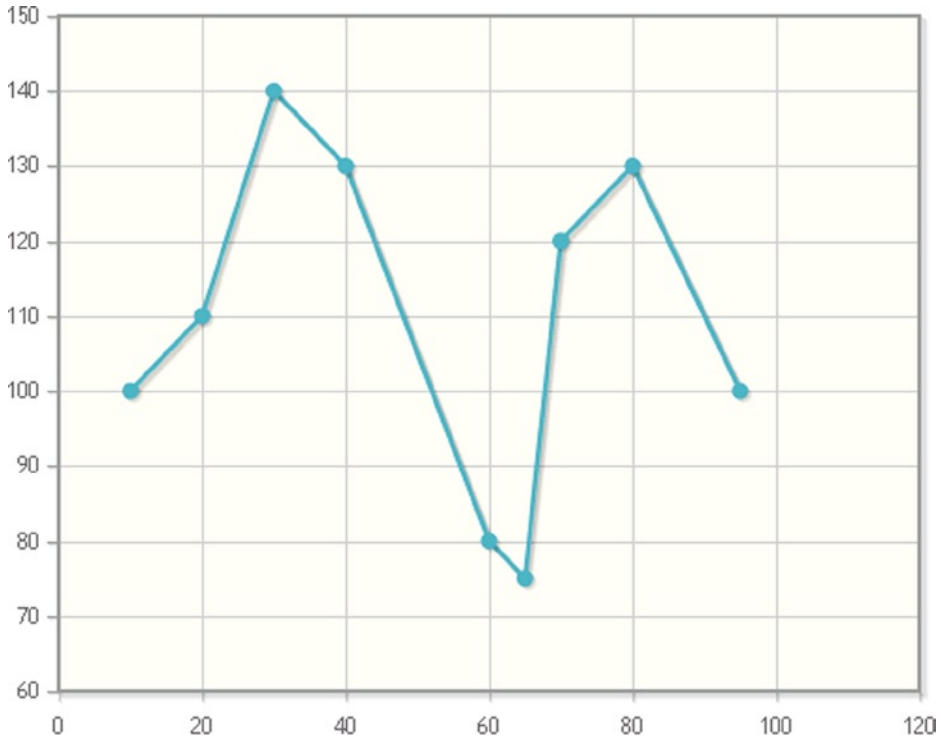


Figure 9-2. *A simple line chart with nonuniformly distributed points on the x axis*

First Steps in the Development of a Line Chart: The Axes

Before looking at the more complex aspects of the line chart in detail, let us first examine the basis on which this kind of chart is represented: the axes. Proper management of the axes is crucial if you want to develop a chart that effects a perfect visualization of data. To this end, you need a good understanding of the modes of action that the jqPlot library offers through the use of specific properties in the `options` object.

Add a Title and Axis Labels

When developing a chart, the first step is to add a title and to manage the axis labels, using the *CanvasAxisLabelRenderer* plug-in.

But, in order to function properly, this plug-in requires another plug-in, one that provides the writing functionality: *CanvasTextRenderer*. With this plug-in, you can render label text directly on canvas elements. This allows you to treat the text like any other graphic element, giving you the ability to rotate the text as you wish. By default the axis label on the y axis is now rotated by 90 degrees, as shown in Figure 9-3.

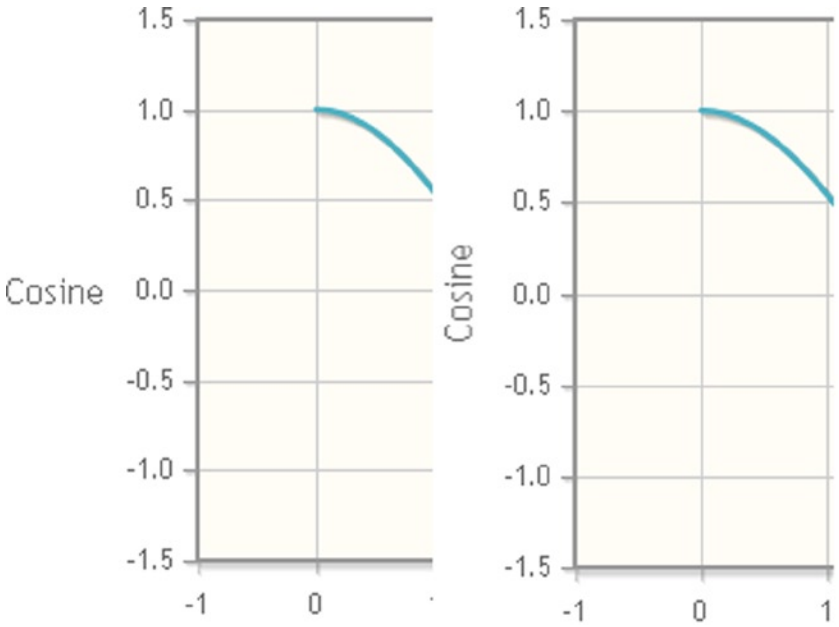


Figure 9-3. *Without including the CanvasAxisLabelRenderer plug-in, the y axis label is horizontal. When the plug-in is included, the y axis label is rotated vertically*

To integrate this new functionality, you need to add the two plug-ins to the basic set of plug-ins:

```
<script type="text/javascript" src="../src/plugins/jqplot.canvasTextRenderer.min.js">
</script>
<script type="text/javascript"
    src="../src/plugins/jqplot.canvasAxisLabelRenderer.min.js"></script>
```

Or, if you prefer to use a content delivery network (CDN) service, you can do so as follows:

```
<script type="text/javascript" src="http://cdn.jsdelivr.net/jqplot/1.0.8/plugins
    /jqplot.canvasTextRenderer.min.js"></script>
<script type="text/javascript" src="http://cdn.jsdelivr.net/jqplot/1.0.8/plugins↵
    /jqplot.canvasAxisLabelRenderer.min.js"></script>
```

Having created the `options` variable, you must then specify some properties inside, as shown in Listing 9-3. You have already seen how to add a title by assigning a string to the `title` object. Then, have to make an explicit call to the `canvasAxisLabelRenderer` object in order to activate its functionality, and by doing so inside the `axesDefaults` object, it will be valid for all axes. To assign the text in both the x axis and y axis labels, you have to set the `label` properties in the `xaxis` and `yaxis` child objects of the `axes` object. Its tree structure will allow you to carry out different changes at the level of each individual axis.

Listing 9-3. ch9_03a.html

```
$(document).ready(function(){
    var data = [[100, 110, 140, 130, 80, 75, 120, 130, 100]] ;
    var options = {
       title: 'My Line Chart',
```

```
        axesDefaults: {
            labelRenderer: $.jqplot.CanvasAxisLabelRenderer
        },
        axes: {
            xaxis: {
                label: "X Axis"
            },
            yaxis: {
                label: "Y Axis"
            }
        }
    };
    $.jqplot ('myChart', data, options);
});
```

Figure 9-4 illustrates the chart the listing code produces.

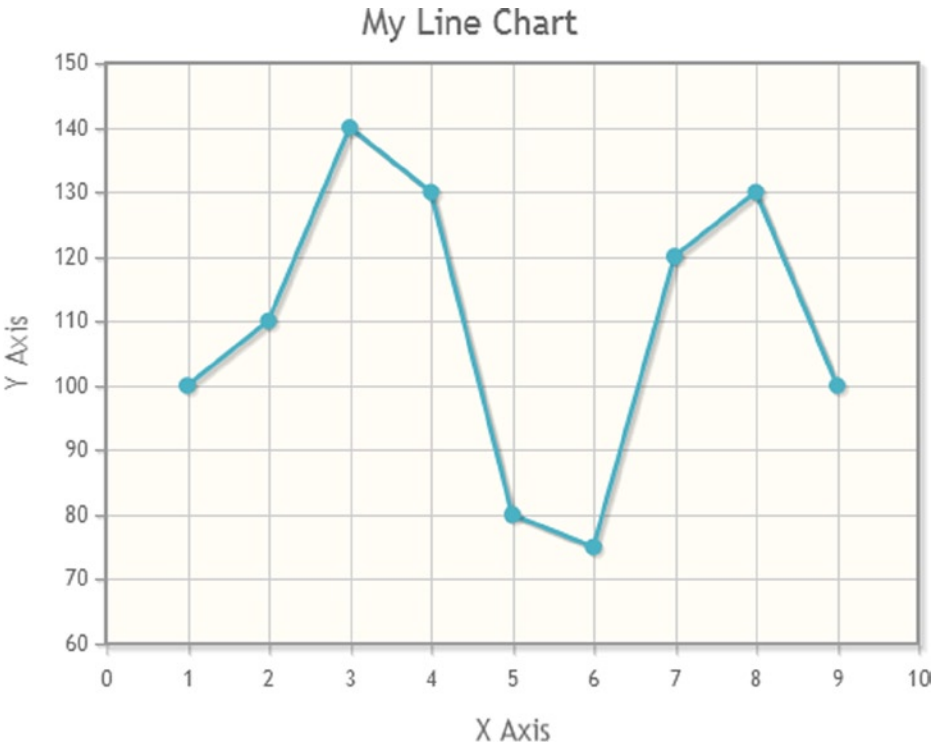


Figure 9-4. *A line chart with the y axis label vertically oriented*

Axis Properties

As with axis labels, there are several properties that can be specified within the `axes` object. For example, looking at the chart (see Figure 9.4), you can see that the line starts from the x value 1, whereas the x axis starts from the value 0, thus leaving an empty space. Another space is seen at the end of the x range (between 9 and 10). If you want to act on

these distances (between the limits of the axes and the end points of your data set), you have to use the pad properties. You apply the padding to extend the range above and below the data bounds. The data range is multiplied by this factor to determine minimum and maximum axis bounds. A value of 0 will be interpreted to mean no padding, and pad will be set to 1. Thus, by adding the pad properties to the xaxis object and setting pad to 1 (see Listing 9-4), you get the chart in Figure 9-5.

Listing 9-4. ch9_03b.html

```
xaxis: {
    label: "X Axis",
    pad: 1
},
```

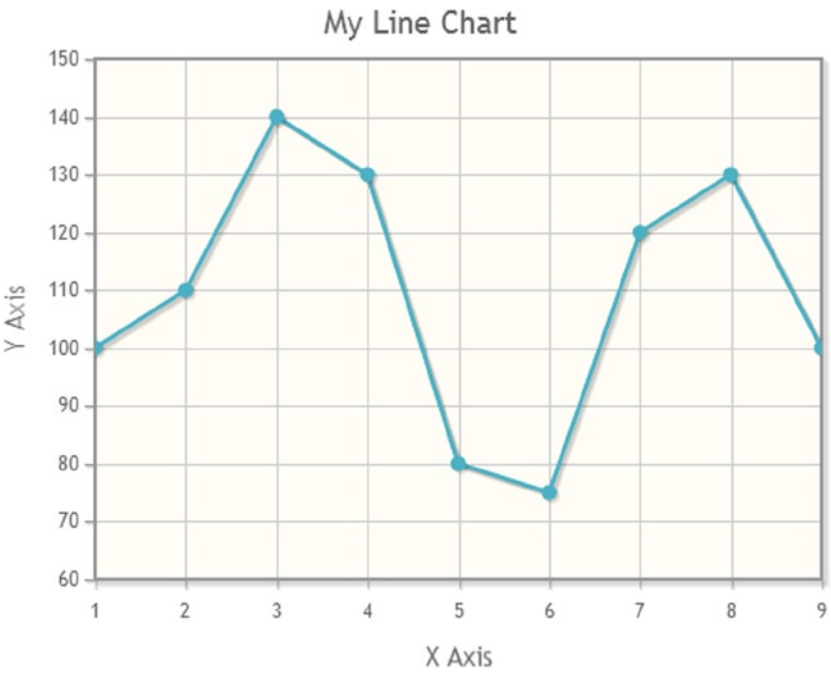


Figure 9-5. *The same line chart as in Figure 9-4, with pad set to 1 on the x axis*

Now, the x axis starts from value 1 and ends with 9, as does the line representing the data series. To better understand the concept of padding, you will now set the pad property to 2 (see Listing 9-5). This means that you want to extend the current range (which is 10) two times. As a result, you will have a chart with an x axis that goes from –4 to 14, as demonstrated in Figure 9-6. This is because jqPlot tends to keep the data in a symmetrical manner, showing it in the middle.

Listing 9-5. ch9_03c.html

```
xaxis: {
label: "X Axis",
pad: 2
},
```

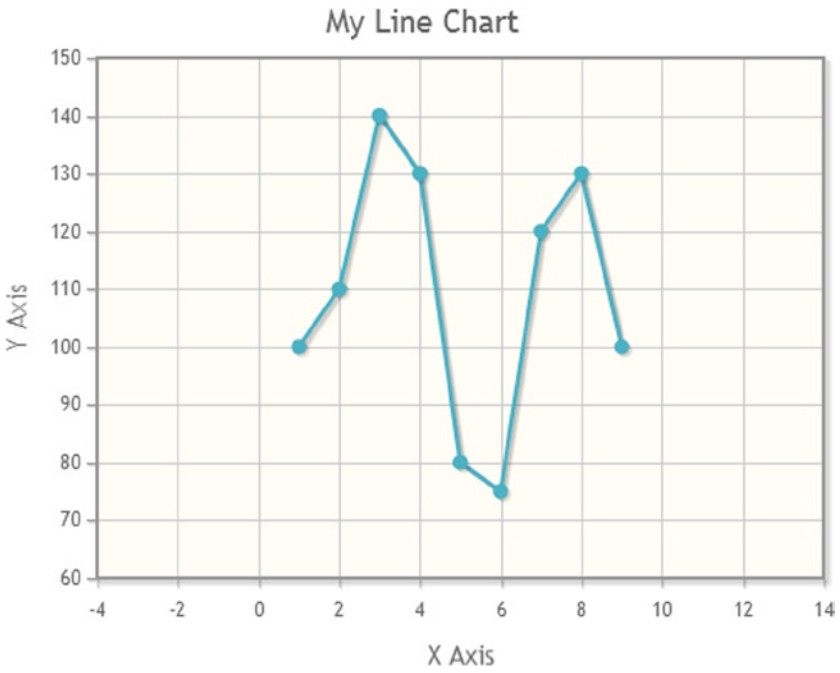


Figure 9-6. *The same line chart, with pad set to 2 on the x axis*

Another way to control the range in which you can display your data is by using the `min` and `max` properties (see Listing 9-6).

Listing 9-6. ch9_03d.html

```
xaxis: {
    label: "X Axis",
    min: 1,
    max: 9
},
```

Figure 9-7 shows the x axis with the new range.

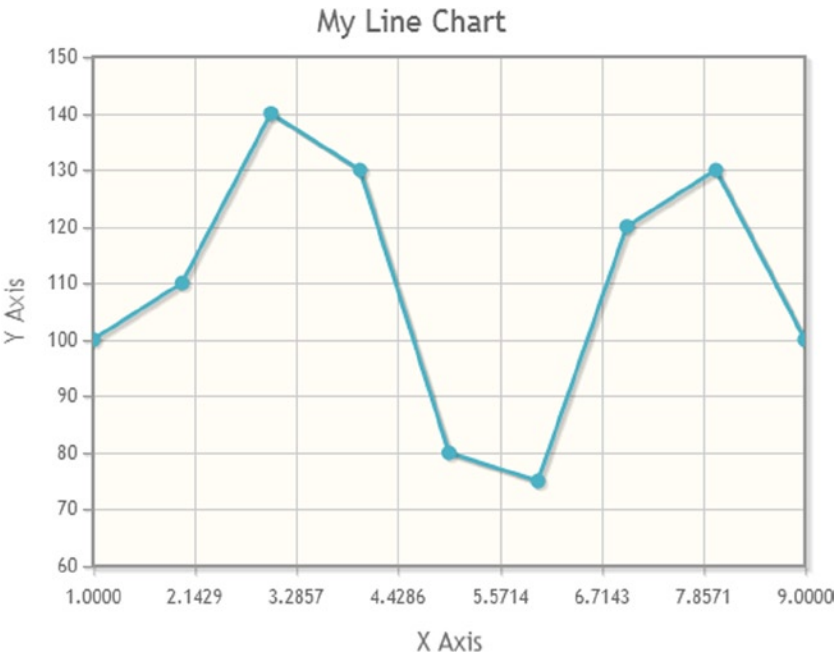


Figure 9-7. *The same line chart, with defined* `max` *and* `min` *on the x axis*

Other useful properties are those that control subdivisions (split axes) and their underlying numeric term: the `ticks` properties. As their use is not limited to simple options under the `axes` object—they are themselves an object and require a renderer plug-in in order to work—their treatment deserves a separate section.

Axes Ticks

A **tick** is a component that shows the value of a tick or grid line in the plot. A tick's behavior in the plot can be specified inside `axes` objects in `options`, and, being an object itself, a tick has several properties that can be set inside the `tickOption` property. For example, you may need to set a specific number of grid lines for each axis. This can be done in different ways. The most simple entails directly specifying the `numberTicks` property (see Listing 9-7). If you set its value to 5, you will get five ticks on the x axis: 0, 3, 6 ,9, and 12 (see Figure 9-8).

Listing 9-7. ch9_03e.html

```
xaxis: {
    label: "X Axis",
    numberTicks: 5
},
```

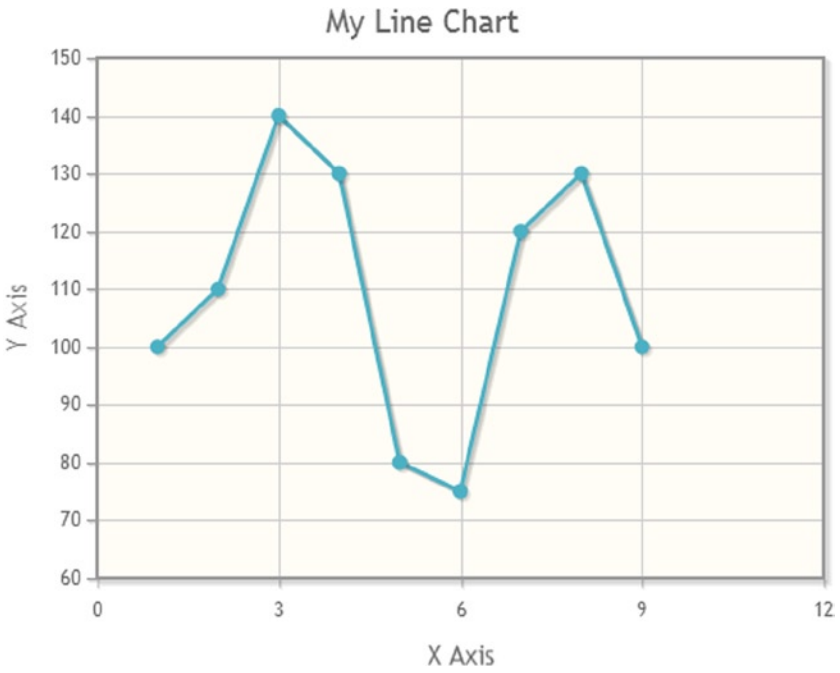


Figure 9-8. *A line chart with a prefixed number of ticks on the x axis*

This can be applied to the y axis, too. In that case, you need to set the same properties in the `yaxis` object. From what you can see, the intervals at which the x axis is divided are uniform, and so the ticks are equidistant. Another way to do this is to define the ticks you want displayed on the chart directly, as shown in Listing 9-8.

Listing 9-8. ch9_03f.html

```
xaxis: {
    label: "X Axis",
    ticks: [0,3,6,9,12]
},
```

This produces the same chart (see Figure 9-9).

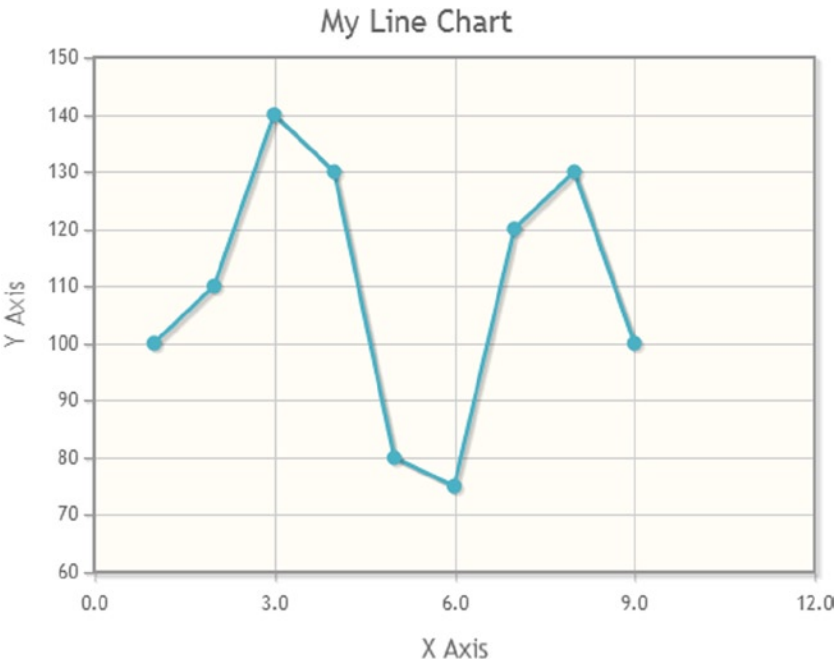


Figure 9-9. *A line chart with directly defined ticks on the x axis*

But, it is generally preferable to use this approach when you want an uneven distribution of ticks along the axis, as in Listing 9-9. The line of the grid will also follow this nonuniformity, as it will be drawn in correspondence with each tick (see Figure 9-10).

Listing 9-9. ch9_03g.html

```
xaxis: {
    label: "X Axis",
    ticks: [1,2,3,7,9]
},
```

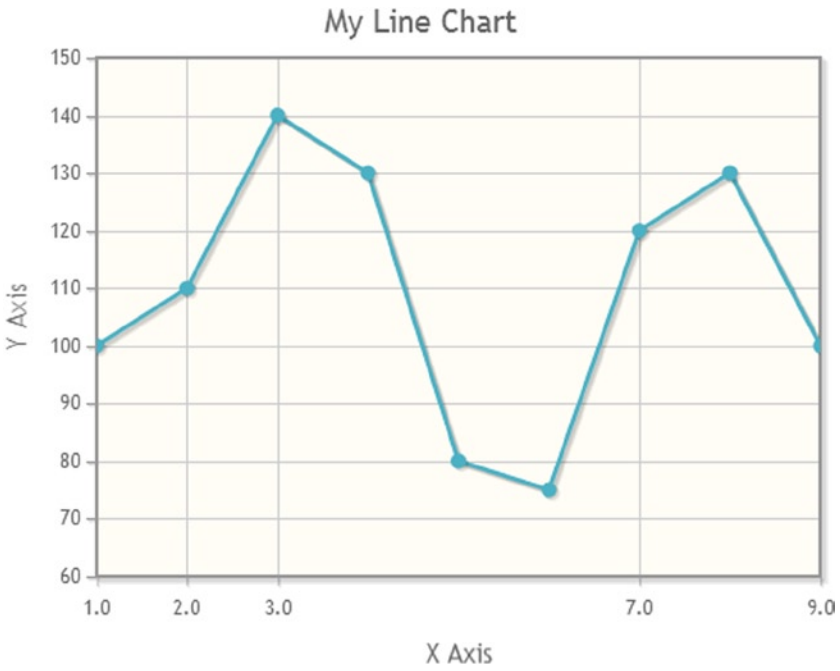


Figure 9-10. *A line chart with nonuniform, prefixed ticks on the x axis*

Ticks are so important in a chart that they have a plug-in that is dedicated specifically to them: *CanvasAxisTickRenderer*.

If you want to create a chart without grid lines while keeping the values on ticks, you can set the `showGridLine` property to `'false'`. Before that, however, you need to include the plug-in in the web page:

```
<link rel="stylesheet" type="text/css" href="../src/jquery.jqplot.min.css" />
<script type="text/javascript"
    src="../src/plugins/jqplot.canvasTextRenderer.min.js"></script>
<script type="text/javascript"
    src="../src/plugins/jqplot.canvasAxisLabelRenderer.min.js"></script>
<script type="text/javascript"
    src="../src/plugins/jqplot.canvasAxisTickRenderer.min.js"></script>
```

Or, if you prefer to use a CDN service, you can do so as follows:

```
<script type="text/javascript" src="http://cdn.jsdelivr.net/jqplot/1.0.8/plugins↩
    /jqplot.canvasTextRenderer.min.js"></script>
<script type="text/javascript" src="http://cdn.jsdelivr.net/jqplot/1.0.8/plugins↩
    /jqplot.canvasAxisLabelRenderer.min.js"></script>
<script type="text/javascript" src="http://cdn.jsdelivr.net/jqplot/1.0.8/plugins↩
    /jqplot.canvasAxisTickRenderer.min.js"></script>
```

You must then make the settings inside the `axesDefaults` object, because you want to hide the grid lines for both axes. Remember to call the plug-in just included with the `tickRenderer` property (see Listing 9-10). Furthermore, you must not forget to delete the **`ticks`** property defined within the **`xaxis`** object.

Listing 9-10. ch9_04a.html

```
axesDefaults: {
    labelRenderer: $.jqplot.CanvasAxisLabelRenderer,
    tickRenderer: $.jqplot.AxisTickRenderer,
    tickOptions: {
       showGridline: false
    }
},
axes: {
    xaxis: {
       label: "X Axis"  //remove the comma here
    },
```

As in Figure 9-11, you get a chart without a grid.

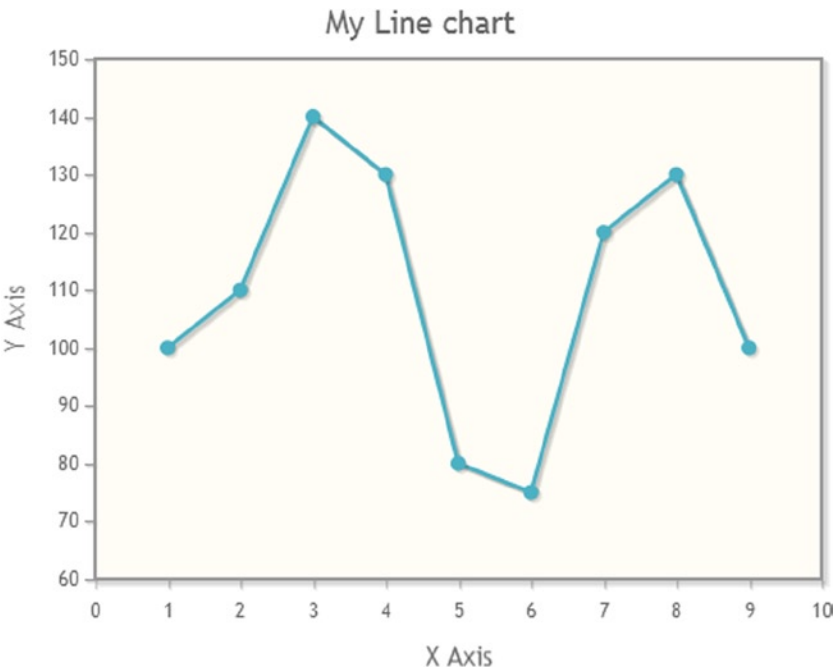


Figure 9-11. *A line chart without grid lines*

Sometimes, you need to hide the grid lines only for one axis, for example, the x axis (see Figure 9-12). In this case, you have to call the renderer inside only the `xaxis` object. In Listing 9-11, you can see the rows of code that must be removed from `axesDefaults` and then written within the `xaxis` object.

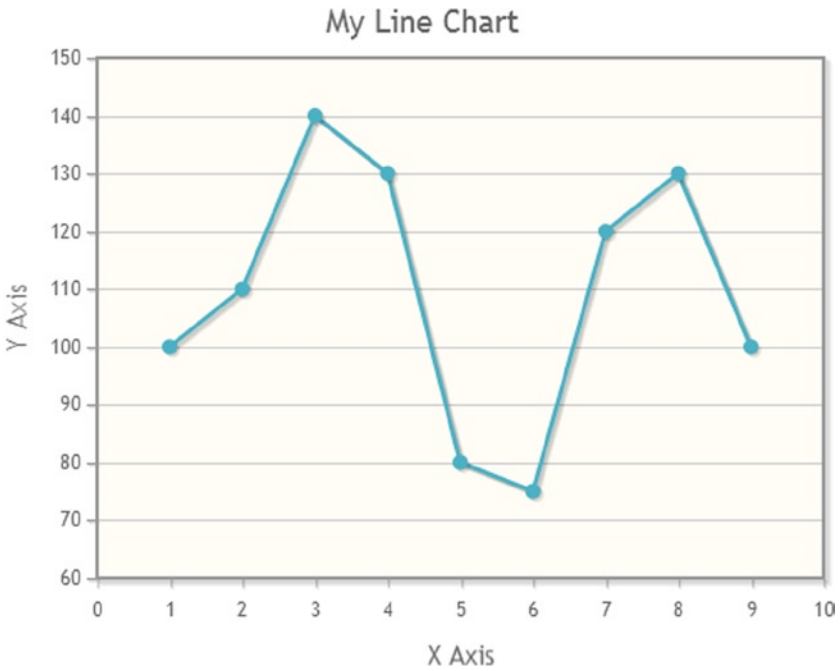


Figure 9-12. *A line chart with only horizontal grid lines*

Listing 9-11. ch9_04b.html

```
axesDefaults: {
    labelRenderer: $.jqplot.CanvasAxisLabelRenderer
    //delete all this lines
    //tickRenderer: $.jqplot.AxisTickRenderer,
    //tickOptions: {
    //showGridline: false
    //}
},
axes: {
    xaxis: {
       label: "X Axis",
       tickRenderer: $.jqplot.AxisTickRenderer,
       tickOptions: {
          showGridline: false
       }
    },
    ...
```

Another possible functionality you may want to add is one that allows you to handle the format of the numeric values as strings. The most common situation in which this could be useful is when you want to show percentage values on the y axis. To accomplish this, you need to add the char `'%'` after the numeric value, as shown in Listing 9-12.

Listing 9-12. ch9_04c.html

```
yaxis: {
    label: "Y Axis",
    tickRenderer: $.jqplot.AxisTickRenderer,
    tickOptions: {
       formatString:'%d%'
    }
}
```

As Figure 9-13 illustrates, the chart now reports percentage values on the y axis.

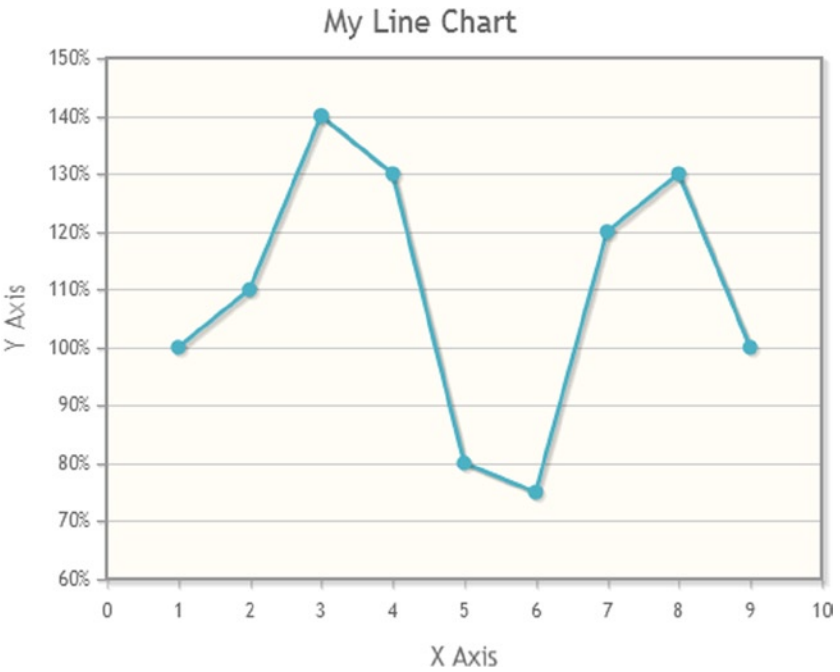


Figure 9-13. *A line chart reporting percentages on the y axis*

Later, you will see other cases in which this kind of string formatting proves to be a very powerful tool (see the section "Handling Date Values").

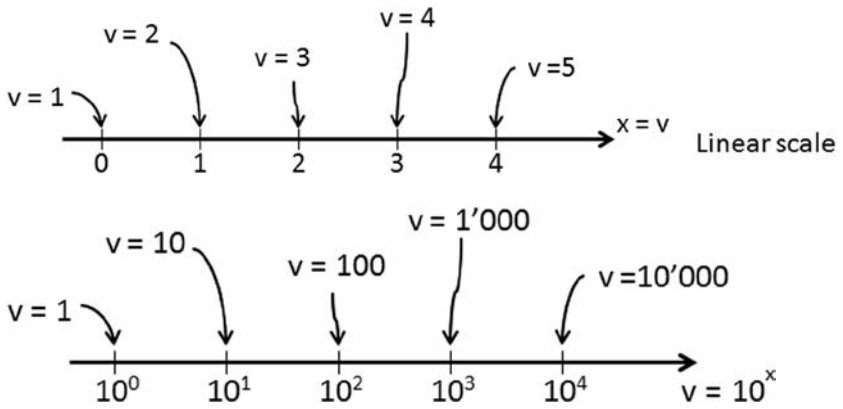
Using the Log Scale

Depending on the trend of the data that you want to represent in a chart, it is sometimes necessary to use a log scale on one, or even both, of the axes. jqPlot supports the log scale, including the *LogAxisRenderer* plug-in in your web page.

```
<script type="text/javascript"
    src="../src/plugins/jqplot.logAxisRenderer.min.js"></script>
```

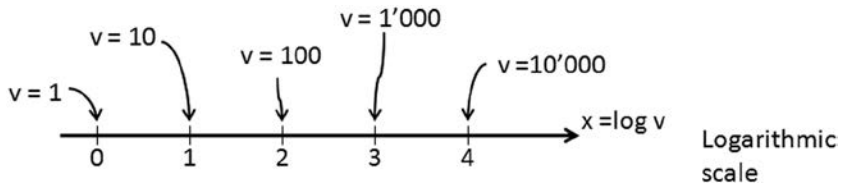
LOG SCALE

The log scale uses intervals corresponding to orders of magnitude (generally ten) rather than a standard linear scale. This allows you to represent a large range of values (v) on an axis.



The logarithm is another way of writing exponentials, and you can use it to separate the exponent (x) and place it on an axis.

$$x = \log_{10} v \Rightarrow v = 10^x$$



For example, an increase of one point on a log scale corresponds to an increase of 10 times that value. Similarly, an increase of two points corresponds to an increase of 100 times that value. And so on.

On the axis on which you want to represent the data in log scale, it is only necessary to add the `renderer` property with the plug-in reference. In this case, you need to create a data array that follows an exponential trend approximately. So, you use the array of [x, y] pairs in Listing 9-13.

Listing 9-13. ch9_11.html

```
var data = [[0,1.2],[10,2.4],[20,5.6],[30,12],[40,23],
           [50,60],[60,120],[70,270],[80,800]];
```

Next, you put the y axis on log scale, as shown in Listing 9-14.

Listing 9-14. ch9_11.html

```
$.jqplot ('myChart', [data],{
    axes:{
    xaxis:{},
    yaxis:{ renderer: $.jqplot.LogAxisRenderer }
    }
});
```

In Figure 9-14, you can see how the data assume, in a semilog scale (log scale on one axis), a shape approximating to a straight line.

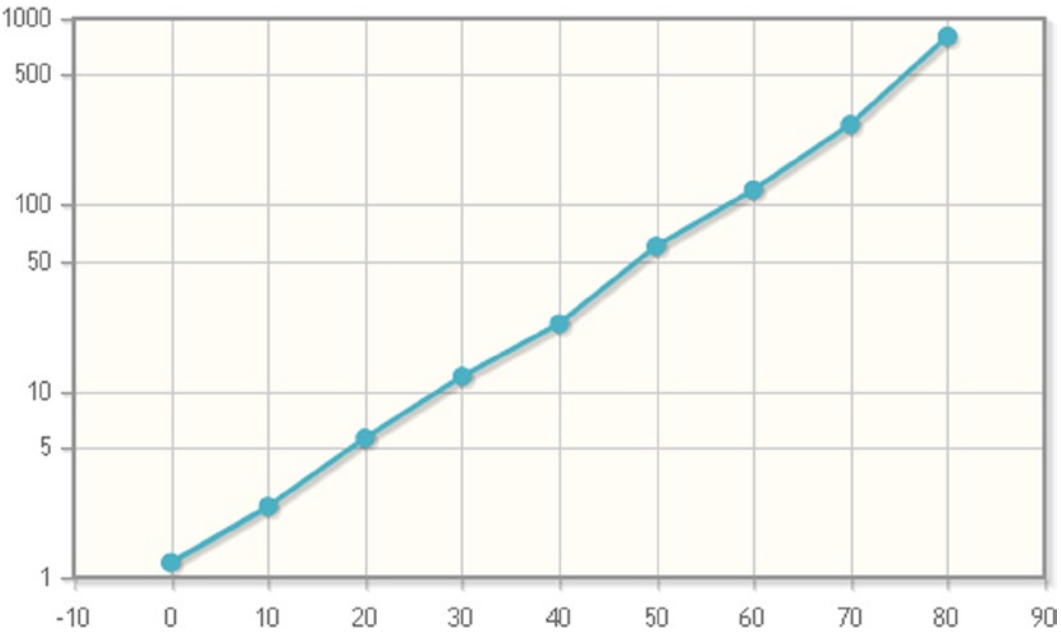


Figure 9-14. *A line chart on a semilog scale on the y axis*

The Multiseries Line Chart

Now that the axes on which you will represent your line chart have been well specified, the time has come to address the multiseries line chart. Typically, you will need to display more than a single series of data in the same chart. Indeed, very often the purpose of a chart is precisely the comparison of different data series.

The jqPlot library provides us with the tools needed to manage multiseries charts. By acting on the patterns, shapes, and colors of lines and markers, it is possible to introduce graphic effects that can aid in the representation of different data series.

Multiple Series of Data

So far, you have been working with only a single set of data. Sometimes, however, you want to represent more than one data set at once. In Chapter 1, you saw that in jqPlot, multiple series are handled in the same way as a single set. Each series must first be defined separately by assigning it to a variable and then combined with the other series in an array. This array is then passed as the second argument to the `jqPlot()` function (see Listing 9-15).

Listing 9-15. ch9_05a.html

```
$(document).ready(function(){
    var data1 = [1, 2, 3, 2, 3, 4];
    var data2 = [3, 4, 5, 6, 5, 7];
    var data3 = [5, 6, 8, 8, 7, 9];
    var data4 = [7, 8, 9, 9, 10, 11];
    var options = {
       title:'Multiple Data Arrays'
    };
    $.jqplot ('myChart', [data1, data2, data3, data4], options);
});
```

Figure 9-15 shows the resulting multiseries chart from Listing 9-15.

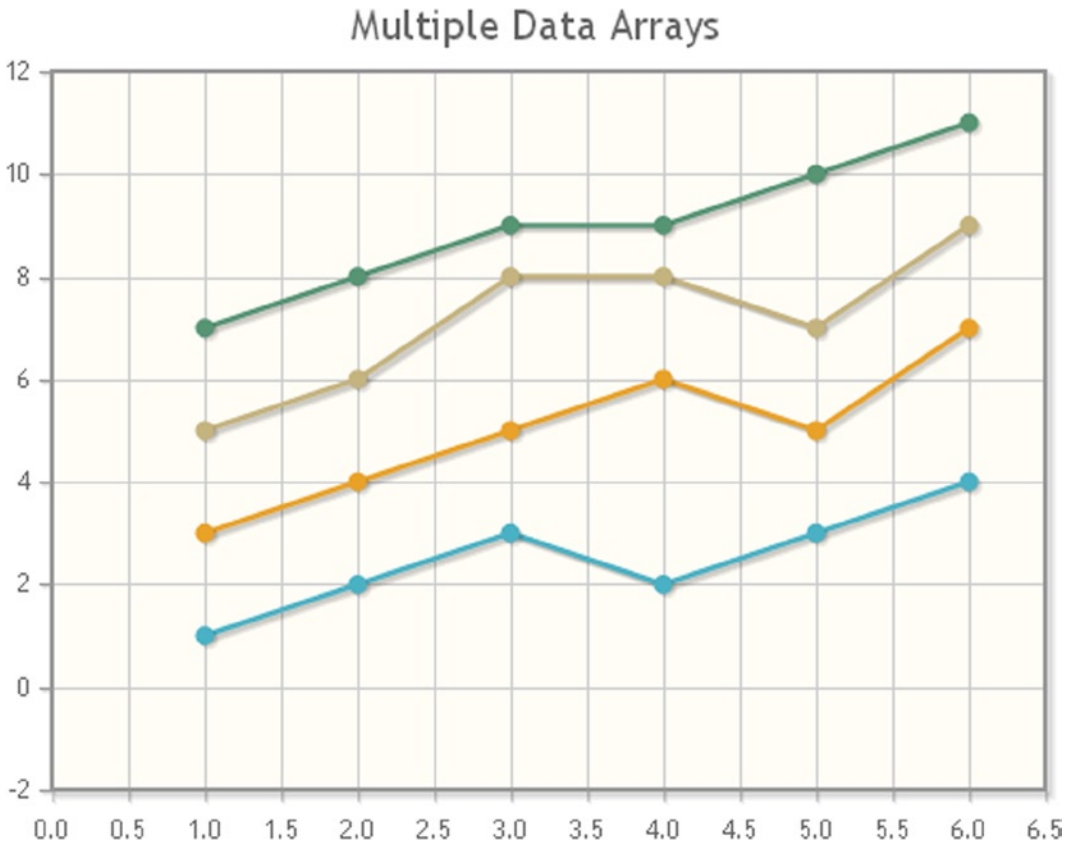


Figure 9-15. *A multiseries line chart*

The system automatically gives each series a different color. This sequence of colors is defined within jqPlot as the default. Here are the colors that jqPlot will assign to the series, in order:

```
seriesColors: [  "#4bb2c5", "#c5b47f", "#EAA228", "#579575",
                 "#839557", "#958c12", "#953579", "#4b5de4",
                 "#d8b83f", "#ff5800", "#0085cc"]
```

These values stand for `'#rrggbb'`, where *rr*, *gg*, and *bb* are the hexadecimal values for red, green, and blue. The browser combines these values to generate all the colors needed for the series.

When there are more than 11 series, jqPlot starts the sequence again from the beginning. If you do not want this or simply need to do things differently, you can define an array with a different sequence of colors in the `seriesColors` property, such as the series given in Listing 9-16. Figure 9-16 shows a variation of gray, but run the example, and see the difference for yourself (the colors ranging from blue to violet).

■ **Note** To check the color codes, I suggest visiting the web site HTML Color Codes (`http://html-color-codes.info`).

Listing 9-16. ch9_05b.html

```
var options = {
    seriesColors: ["#105567","#805567","#bb5567","#ff5567"],
    title:'Multiple Data Arrays'
};
```

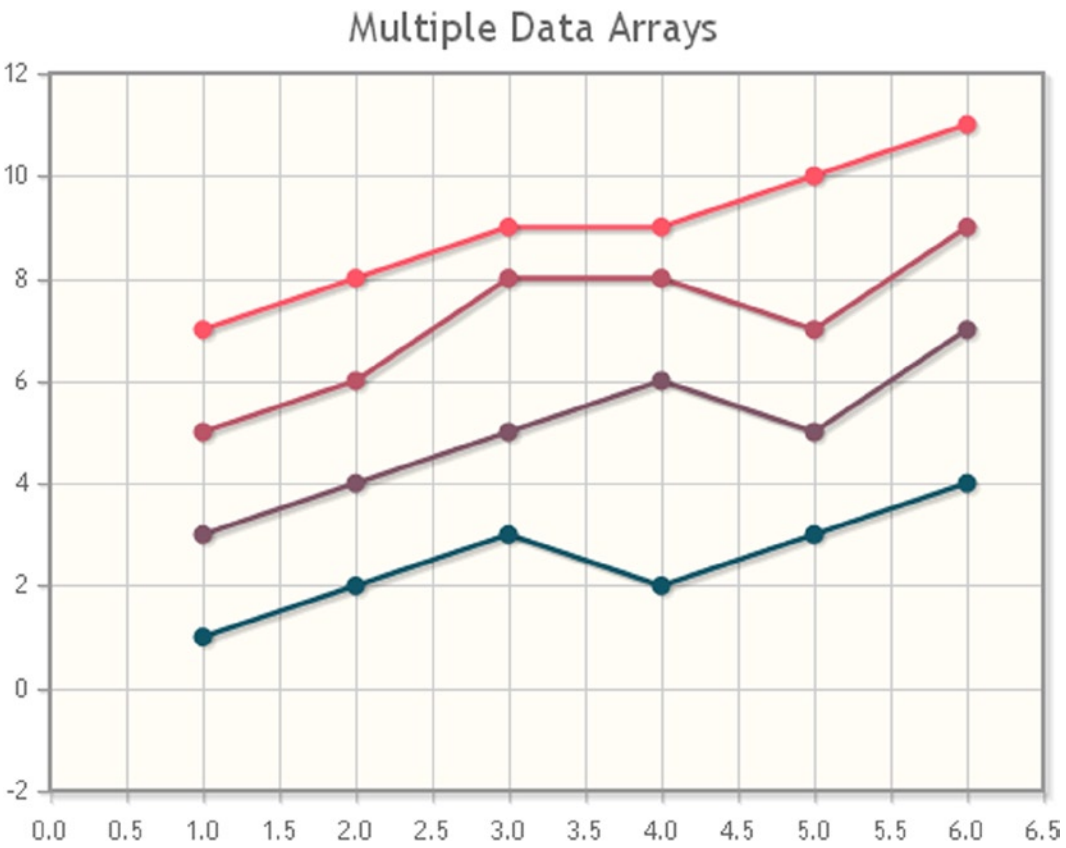


Figure 9-16. *A multiseries line chart with a customized color set*

You can also attribute a specific color, using two functions: `rgba(r,g,b,a)` and `rgb(r,g,b)`. You insert these functions directly in each value of the array to be allocated to the `seriesColors` property, as shown in Listing 9-17.

Listing 9-17. ch9_05c.html

```
seriesColors: ["rgba(16,85,103,0.2)", "rgba(128,85,103,0.6)",
               "rgb(187,85,103)", "rgb(250,85,103)"],
```

Whereas you have been specifying colors through the combination of red, green, and blue light required to achieve a given color, with the `rgba()` function, a new variable, `a`, is introduced. This `a` (for "alpha") stands for the level of opacity/transparency of a color. As Figure 9-17 demonstrates, defining low alpha values lets you see what lies behind the colored object.

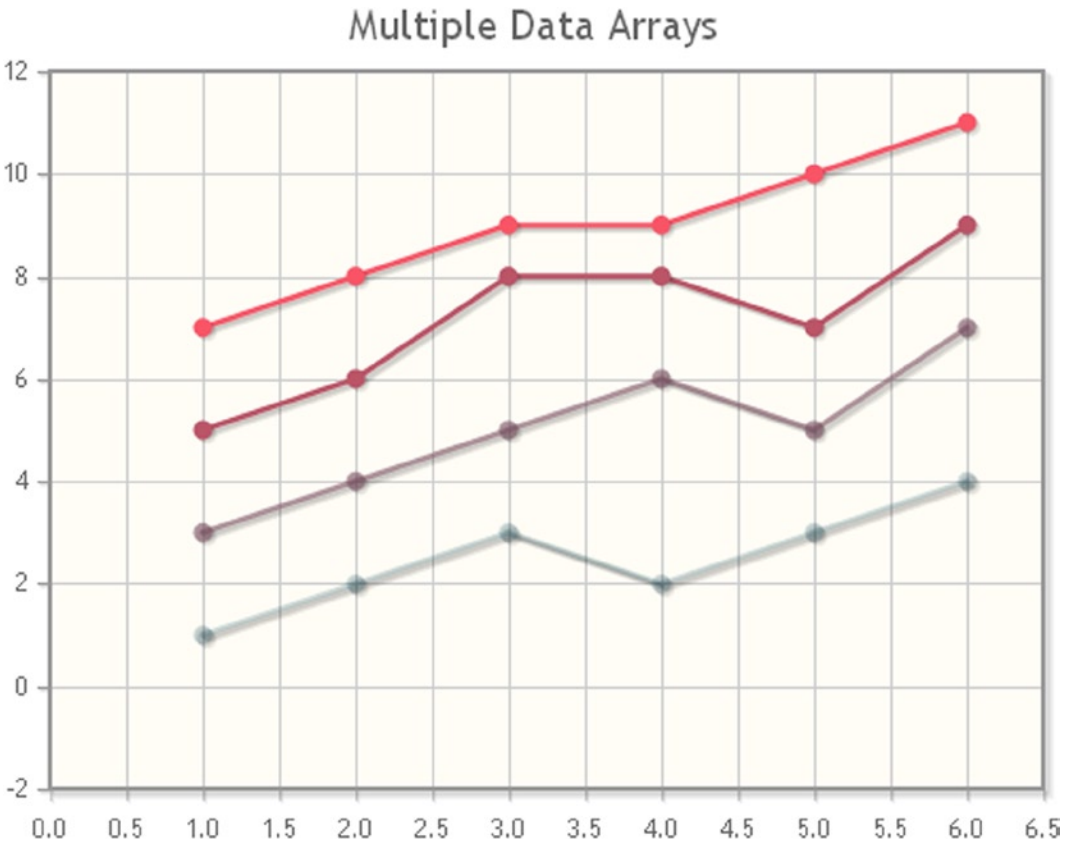


Figure 9-17. *A multiseries line chart with different levels of transparency*

Smooth-Line Chart

In addition to choosing whether to represent dot markers and the straight lines linking them, often you will decide that you want to get a smooth curve progress, as presented in Figure 9-18. This can be done simply by using the `smooth` property and setting it to `'true'` (see Listing 9-18).

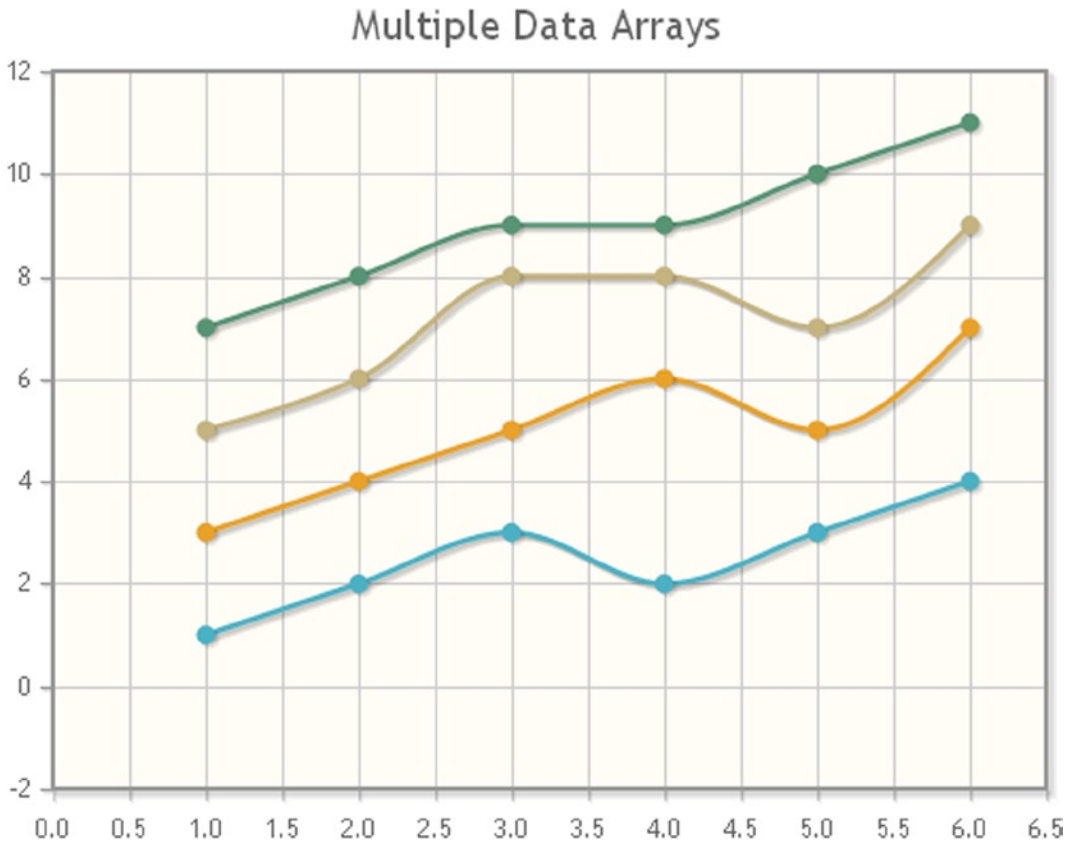


Figure 9-18. *A multiseries line chart with smoothed lines*

Listing 9-18. ch9_06.html

```
$(document).ready(function(){
    var data1 = [1, 2, 3, 2, 3, 4];
    var data2 = [3, 4, 5, 6, 5, 7];
    var data3 = [5, 6, 8, 8, 7, 9];
    var data4 = [7, 8, 9, 9, 10, 11];
    var options = {
       title:'Multiple Data Arrays',
       seriesDefaults: {
          rendererOptions: {
             smooth: true
          }
       }
    };
    $.jqplot ('myChart', [data1, data2, data3, data4], options);
});
```

Line and Marker Style

Another key aspect that you need to take into account while designing your line chart is how lines and markers are displayed. You can represent a chart using a line, a sequence of markers, or both. By default, jqPlot shows each series with dot markers for every point corresponding to the [x, y] pairs and a line joining them in sequence.

All this can be controlled using two key properties belonging to the series objects: linePattern and lineWidth; while adding the `markerOptions` property, it is also possible to act on two other properties affecting marker components: style and size. Listing 9-19 is an example of these settings.

Listing 9-19. ch9_07a.html

```
$(document).ready(function(){
    var data1 = [1, 2, 3, 2, 3, 4];
    var data2 = [3, 4, 5, 6, 5, 7];
    var data3 = [5, 6, 8, 9, 7, 9];
    var data4 = [7, 8, 9, 11, 10, 11];
    var options = {
       title: 'Multiple Data Arrays',
       series:[{
          linePattern: 'dashed',
          lineWidth:2,
          markerOptions: { style: 'diamond' }
       },
       {
          showLine:false,
          markerOptions: { size: 7, style: 'x' }
       },
       {
          markerOptions: { style: 'circle' }
       },
       {
          lineWidth:5,
          linePattern: 'dotted',
          markerOptions: { style: 'filledSquare', size: 10 }
       }]
    }
    $.jqplot ('myChart', [data1, data2, data3, data4], options);
});
```

Figure 9-19 illustrates the result of the settings in Listing 9-19.

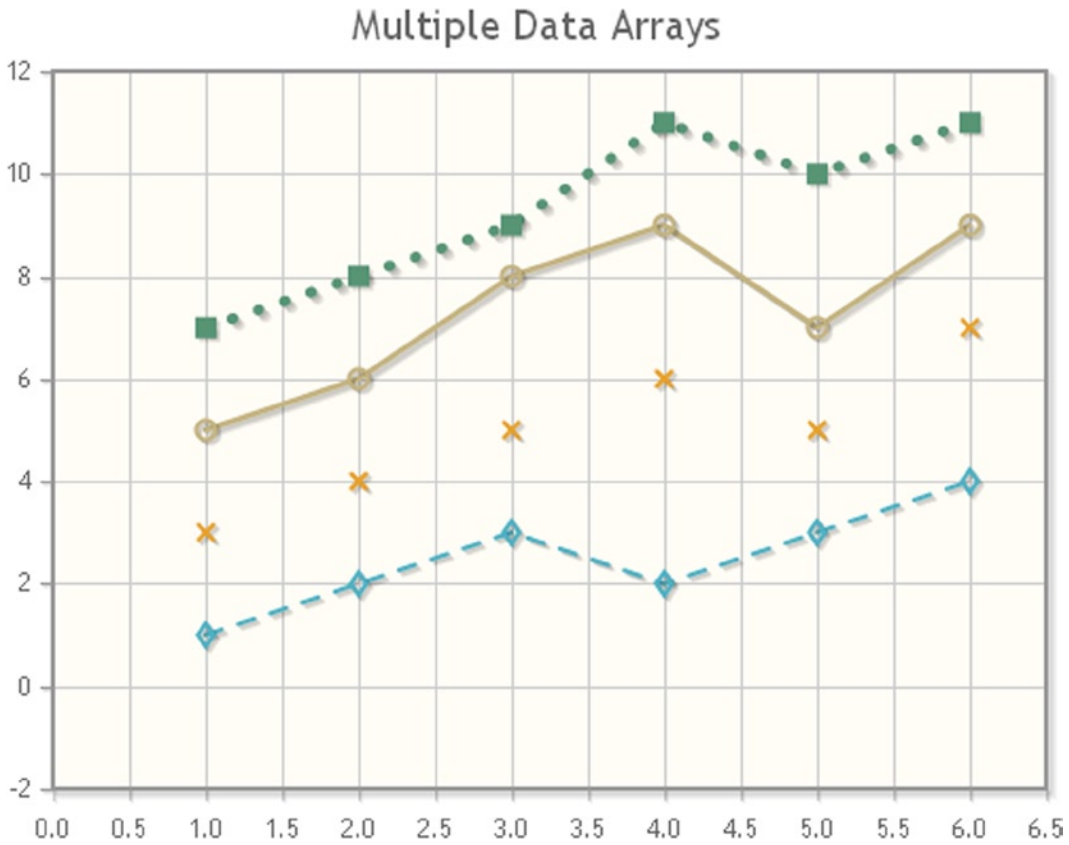


Figure 9-19. *In a line chart it is possible to set different markers and patterns*

The lines on a chart can be drawn as solid, dashed, or dotted with the `linePattern` property. By default every line drawn is solid, so if you want a line to have a different style, it is necessary to specify it in `options`. You saw in Listing 9-19 that it is possible to set the `linePattern` property to `'dotted'` or `'dashed'` in order to obtain a dotted or dashed line, respectively. In Listing 9-20, you can see that it is also possible to obtain a customized line pattern, defining the format as an array ([dash length, gap length, and so on]). A line looks best when the array assigned to the `linePattern` property has an even number of elements, such that the line begins with a dash and ends with a gap. The `linePattern` property can also create a customized pattern, using a shorthand string notation of dash (`-`) and dot (`.`) characters. Listing 9-20 provides examples.

Listing 9-20. ch9_07b.html

```
var options = {
    title: 'Multiple Data Arrays',
    seriesDefaults: {
       showMarker: false
    },
    series: [{ linePattern: 'dashed'},
             { linePattern: 'dotted'},
             { linePattern: [4, 3, 1, 3, 1, 3]},
             { linePattern: '-.'}]
};
```

Figure 9-20 shows the examples of customized line patterns used in Listing 9-20.

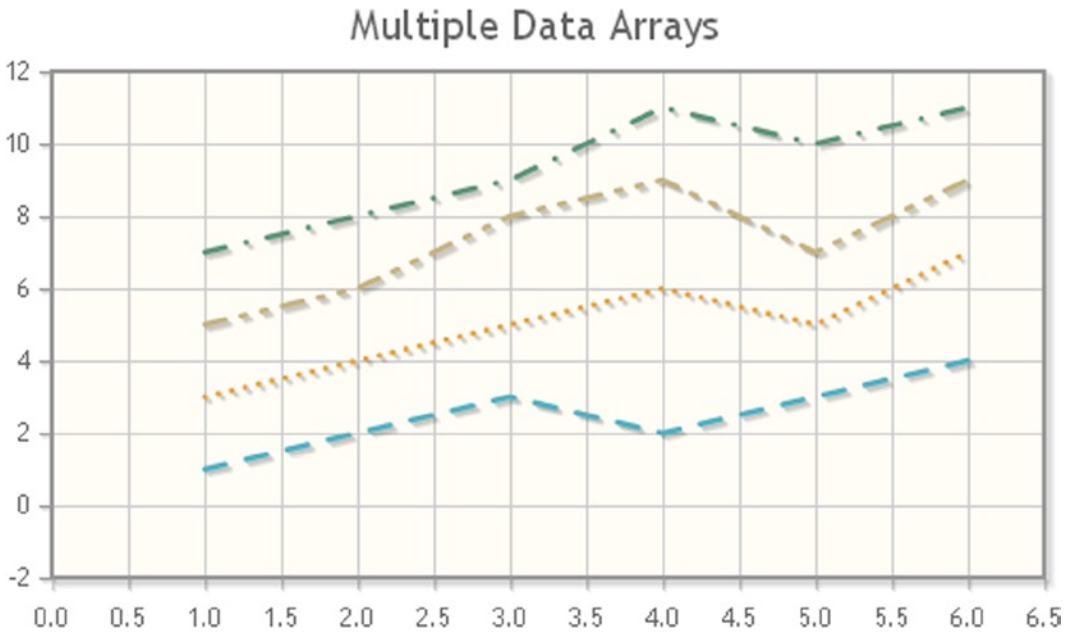


Figure 9-20. *A multiseries line chart with different patterns*

Animated Charts

When you load your web page in the browser, you will note that the chart is drawn almost instantaneously. You can slow down the drawing speed, adjusting it to your preference; a slower speed gives a floating effect to the chart while its elements are being drawn (see Listing 9-21).

Listing 9-21. ch9_23.html

```
var options = {
    title: 'Multiple Data Arrays',
    seriesDefaults: {
       showMarker: false,
       rendererOptions: {
          smooth: true,
          animation: { show: true  }
       }
    }
};
```

Figure 9-21 shows the sequence in which the chart is drawn, giving it an animated look.

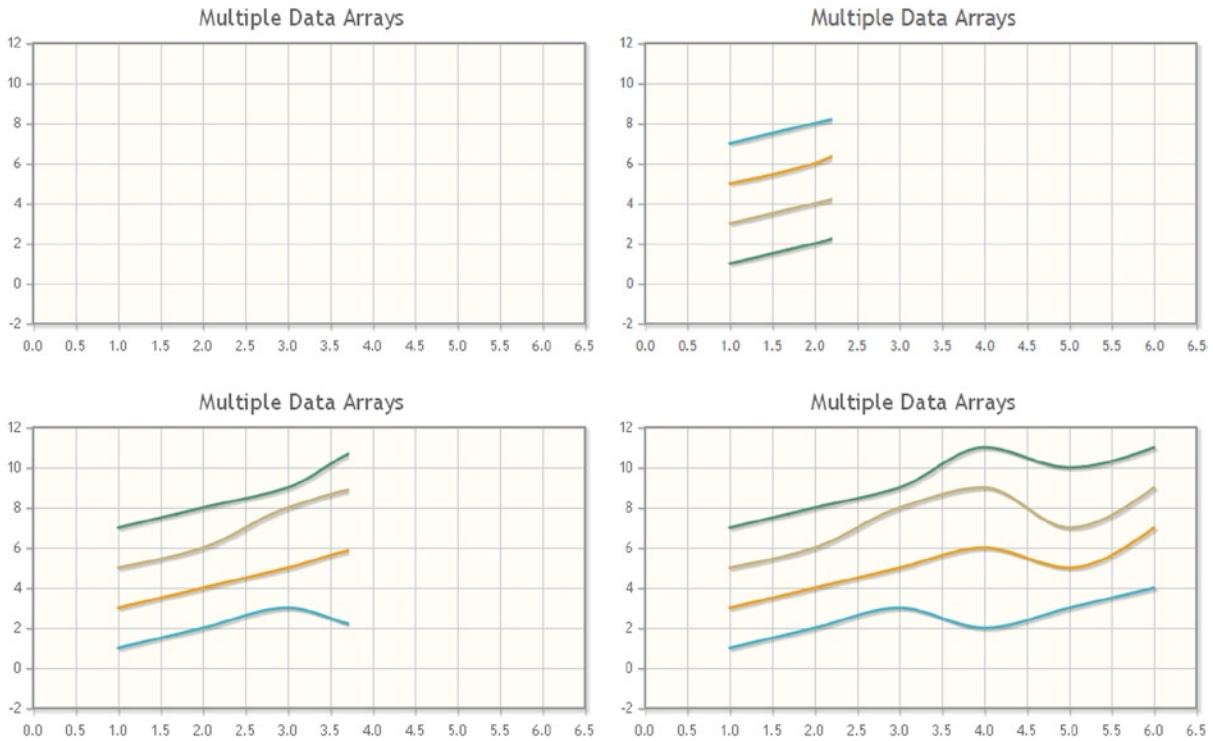


Figure 9-21. *An animated multiseries line chart*

More Than One y Axis

jqPlot supports multiple y axes in relation to the same x axis. This can be useful when, in a single chart, you want to display different series distributed on different y scales, but with the same x values. In such cases, it is wise to set the y axes with the color of the corresponding series so that you can determine the correct y value of any given point. As input data, let us create three data arrays containing the same x values but with y values distributed on different ranges, as shown in Listing 9-22. Using the same x values is not mandatory, but it is smart to do so.

Listing 9-22. ch9_12.html

```
var data1 = [[10, 200], [20, 230], [30, 214], [40, 212], [50, 225], [60, 234]];
var data2 = [[10, 455], [20, 470], [30, 465], [40, 432], [50, 455], [60, 464]];
var data3 = [[10, 40], [20, 60], [30, 54], [40, 52], [50, 65], [60, 54]];
```

It is very important to specify the correct range of values for each y axis in order to be able to compare the different series of values easily (see Listing 9-23). In the `series` object, you need to specify explicitly three values, each of which assigns a series to a different y axis. If you want to keep the default setting for a particular series, that is, representation along the default y axis, you must still assign an empty object {} in the position corresponding to that series. In fact, in this example the first element of the `series` array is just an empty object {}.

In addition, you need to set the `useSeriesColor` property for the `axesDefaults` object to `'true'`. In so doing, jqPlot will assign the color of the series to the corresponding y axis. Thus, by using three default colors, you will get light blue for the first series, orange for the second, and gray-brown for the third.

Listing 9-23. ch9_12.html

```
var options = {
    series:[
      {},
      {yaxis: 'y2axis'},
      {yaxis: 'y3axis'}
    ],
    axesDefaults:{useSeriesColor: true},
    axes:{
       xaxis: {min: 0, max: 70},
       yaxis: {min: 190, max: 240},
       y2axis: {min: 430, max: 480},
       y3axis: {min: 35, max: 80}
    }
};
$.jqplot ('myChart', [data1, data2, data3], options);
```

Figure 9-22 presents the three series, each represented in relation to the values of its y axis. The axes are shown here in different grayscale shades, but they actually assume the colors corresponding to the related series.

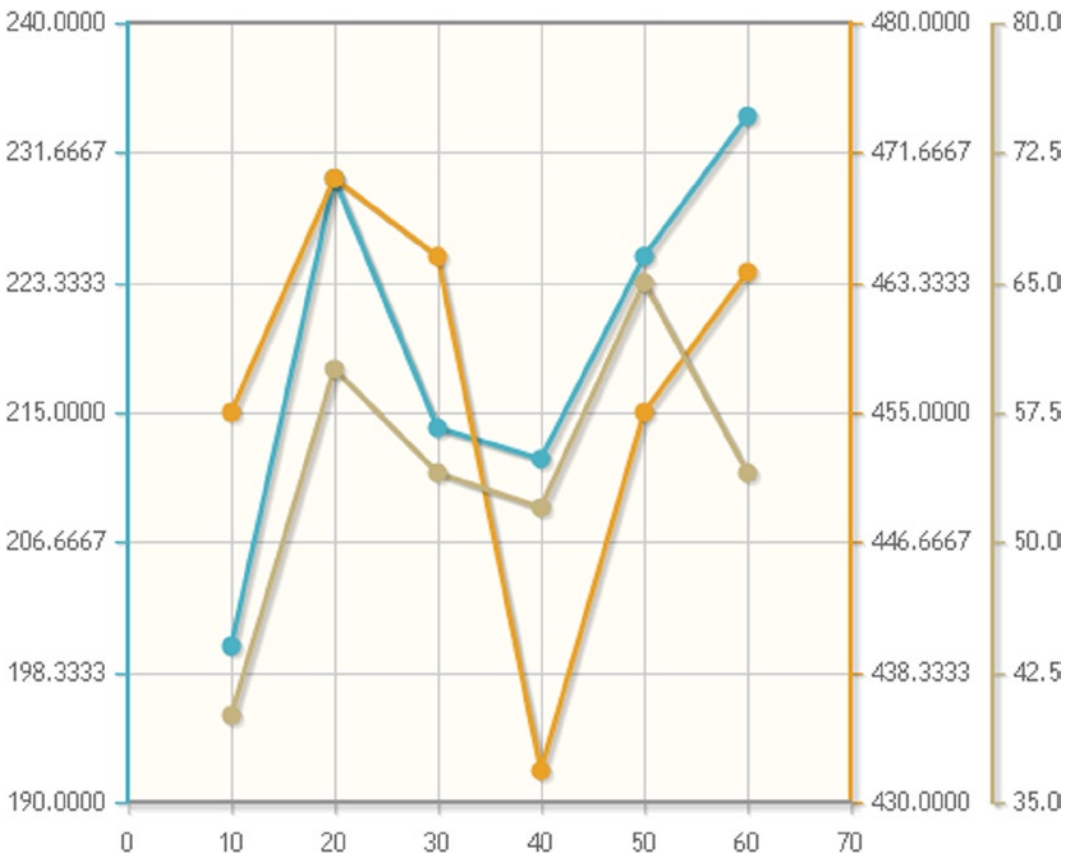


Figure 9-22. *A multiseries line chart with multiple y axes*

Data with JavaScript

As discussed previously, it is preferable to define data arrays separately and outside the `jqPlot` function. You have seen how to create an array containing numeric values that are either y values or [x, y] pairs. Yet, because jqPlot belongs to the world of JavaScript, there is another approach that often proves to be very useful: generating data series through JavaScript methods.

Generating Data, Using Math Functions

The jqPlot library is based on JavaScript, and, as with all programming languages, it allows you to implement functions that generate sequences of values to use as input data. For example, Listing 9-24 takes three of the most used and best-known mathematical functions (sine, cosine, power) and creates an array of data through them.

Listing 9-24. ch9_08a.html

```
$(document).ready(function(){
    var options = {
       title:'Math function Arrays'
    };

    varcosPoints = [];
    for (vari=0; i< 2 * Math.PI; i += 0.1){
       cosPoints.push([i, Math.cos(i)]);
    }

    varsinPoints = [];
    for (vari=0; i< 2 * Math.PI; i += 0.1){
       sinPoints.push([i, 2 * Math.sin(i-.8)]);
    }

    varpowPoints = [];
    for (vari=0; i< 2 * Math.PI; i += 0.1) {
       powPoints.push([i, 2.5 + Math.pow(i/4, 2)]);
    }

    $.jqplot ('myChart', [cosPoints, sinPoints, powPoints], options);
});
```

Figure 9-23 illustrates how the points generated by the three functions in the listing form, on a line chart, the characteristic trends of the three mathematical functions.

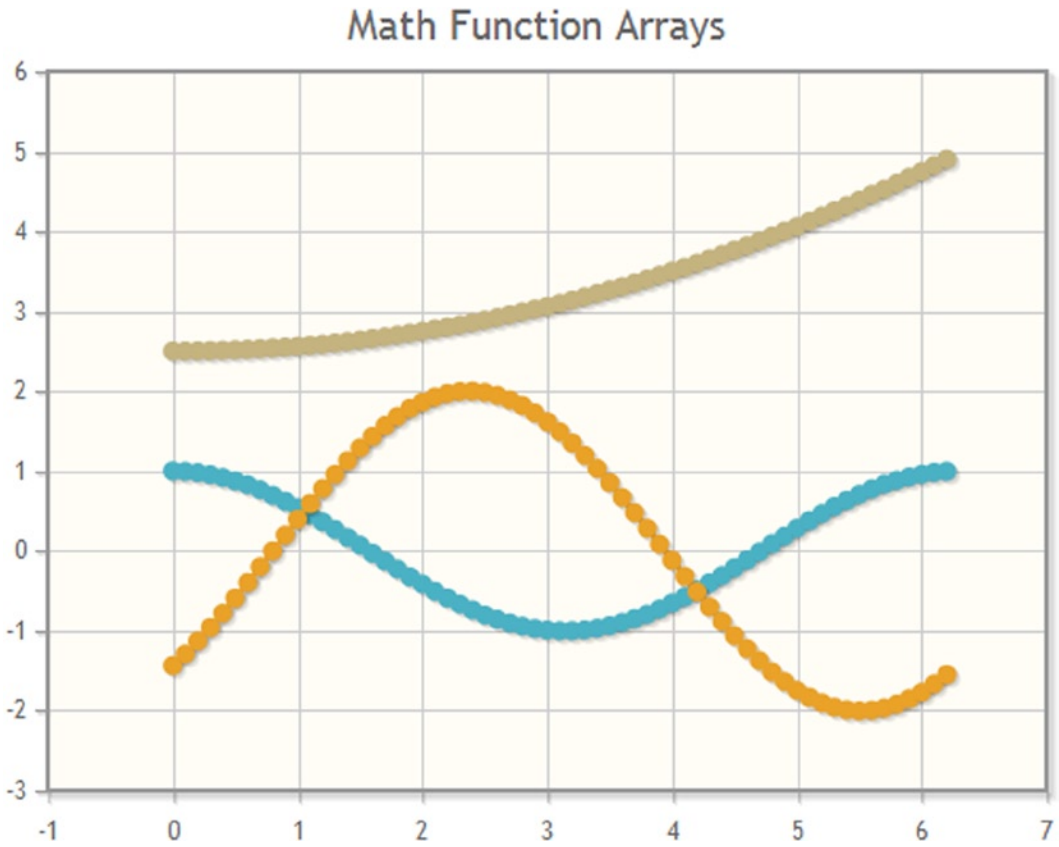


Figure 9-23. *A line chart reporting three different series of data generated from mathematical functions*

Because this is a function with a high density of points, and because the objective here is to highlight trends, it is best not to display the marker points (see Figure 9-24). It is also preferable to enable smoothing in `options`, as shown in Listing 9-25.

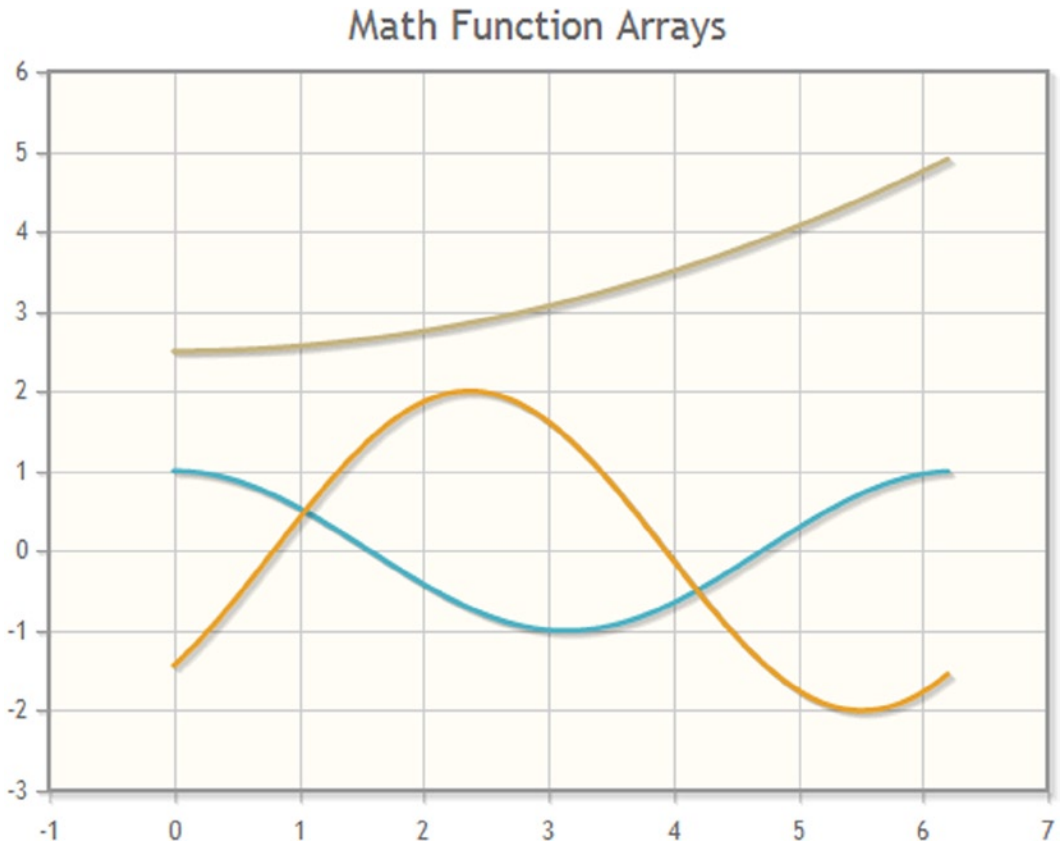


Figure 9-24. *The same line chart, but rendered more legibly*

Listing 9-25. ch9_08b.html

```
var options = {
    title: 'Math function Arrays',
    seriesDefaults: {
       rendererOptions: {
          smooth: true
       },
       markerOptions: { show: false }
    }
};
```

Generating Random Data

You have just seen how to generate input data by using mathematical functions. Similarly, it is sometimes necessary to generate random data. For instance, let us say you have just finished writing your jqPlot chart and would like to try inputting dummy data. To this end, the use randomly generated data is best. The function in Listing 9-26 generates random data, with every point generated according to the value of the previous one. At each step, the new value is determined by a random number that is added to or subtracted from the preceding number. This results in a continuous series of data, starting from a value passed as an argument to the function.

Listing 9-26. ch9_09.html

```
function generateRandomData(npts, start, delta) {
    var data = [];
    if (delta == null) {
       delta = start;
       start = (Math.random() - 0.5) * 2 * delta;
    }
    for (j=0; j<npts; j++) {
       data.push([j, start]);
       start += (Math.random() - 0.5) * 2 * delta;
    }
    return data;
}
```

You are using three arguments: `npts` is the number of points to generate, `start` is the starting value, and `delta` is the maximum value to add or subtract randomly at every step. The function returns an array that will be passed as input data to the chart. You can define it externally:

```
var data = generateRandomData(30, 100, 1);
$.jqplot('myChart', [data]);
```

Or, you can pass it directly:

```
$.jqplot ('myChart', [makeContinuousData(30, 100, 1)]);
```

As a result, you get a chart like the one in Figure 9-25 (it will be different every time).

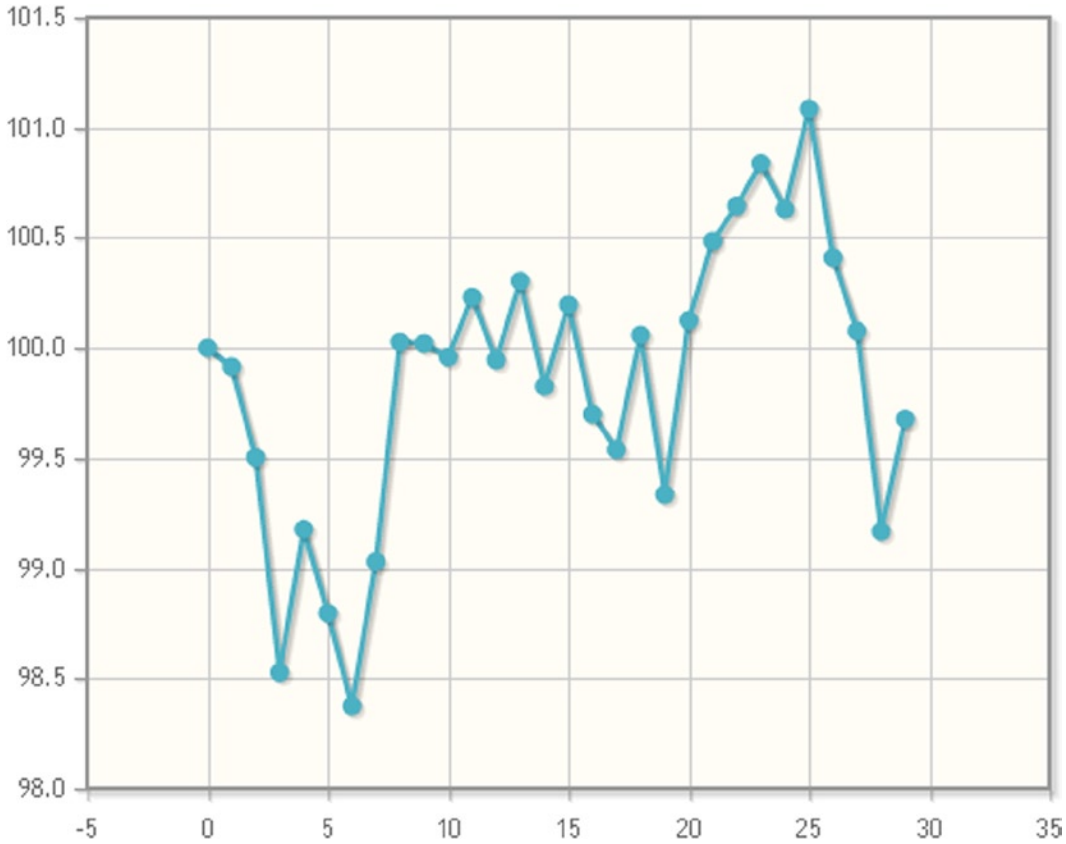


Figure 9-25. *A line chart representing a random series of data*

Handling Date Values

A kind of value that is used, especially in other charts (e.g., bar charts), is date type. These specialized values are not so easy to deal with, and jqPlot has a plug-in for them: *DateAxisRenderer*. This plug-in expands JavaScript's native date-handling capabilities, allowing you to represent date values in any unambiguous form, not just in milliseconds.

The *DateAxisRenderer* Plug-in

A date can be represented in many ways, and its format varies, depending on country and use. A date consists of day, month, and year indicators. These can be ordered differently, and with one, two, or four digits; or, you may even want to use only one or two of the indicators (e.g., month, year). Furthermore, various characters act as separators. Let us take, for example, 04/07/2012: "4" stands for the fourth month (April), "7" is the seventh day of the month, and "2012" is the year. Such a date can be shown in numerous ways: `'07/04/2012'`, `'07/04/12'`, `'04/07/12'`, `'7-Apr-12'`, `'7-Apr'`, `'Apr-12'`, `'7 April'`, `'2012'`, and so on.

The standard format for date values is as follows:

```
'YYYY-MM-DD HH:MM<PM or AM>'
```

This string contains all the necessary information—a bit too much, perhaps. In fact, you will often require only a part of the date information: sometimes, you may need to report only day and month, or, if you refer to time, you may need to handle only hours and minutes, and so on.

Once the *DateAxisRenderer* plug-in is included, jqPlot accepts almost any recognizable value. After the value has been internally parsed, it will be rendered on the axis on which you made the call to the plug-in, represented in the format specified in `tickOptions.formatString`.

Table 9-1 shows the acceptable format codes.

Table 9-1. *Date and Time Formats Accepted by jqPlot*

Code	Result	Description
Years		
%Y	2008	Four-digit year
%y	08	Two-digit year
Months		
%m	09	Two-digit month
%#m	9	One- or two-digit month
%B	September	Full month name
%b	Sep	Abbreviated month name
Days		
%d	05	Two-digit day of month
%#d	5	One- or two-digit day of month
%e	5	One- or two-digit day of month
%A	Sunday	Full name of day of the week
%a	Sun	Abbreviated name of day of the week
%w	0	Number of day of the week (0 = Sunday, 6 = Saturday)
%o	th	Ordinal suffix string following day of the month
Hours		
%H	23	Hours in 24-hour format (two digits)
%#H	3	Hours in 24-hour integer format (one or two digits)
%I	11	Hours in 12-hour format (two digits)
%#I	3	Hours in 12-hour integer format (one or two digits)
%p	PM	AM or PM
Minutes		
%M	09	Minutes (two digits)
%#M	9	Minutes (one or two digits)

(*continued*)

Table 9-1. (*continued*)

Code	Result	Description
Seconds		
%S	02	Seconds (two digits)
%#S	2	Seconds (one or two digits)
%s	1206567625723	Unix timestamp (seconds past 1970-01-01 00:00:00)
Milliseconds		
%N	008	Milliseconds (three digits)
%#N	8	Milliseconds (one to three digits)
Time zone		
%O	360	Difference in minutes between local time and Greenwich mean time (GMT)
%Z	Mountain Standard Time (MST)	Name of time zone, as reported by browser
%G	–06:00	Hours and minutes between GMT
Shortcuts		
%F	2008-03-26	%Y-%m-%d
%T	05:06:30	%H:%M:%S
%X	05:06:30	%H:%M:%S
%x	03/26/08	%m/%d/%y
%D	03/26/08	%m/%d/%y
%#c	Wed Mar 26 15:31:00 2008	%a %b %e %H:%M:%S %Y
%v	3-Sep-2008	%e-%b-%Y
%R	15:31	%H:%M
%r	3:31:00 PM	%I:%M:%S %p
Characters		
%n	\n	New line
%t	\t	Tab
%%	%	Percent symbol

To get a clearer idea of how jqPlot handles date values, let us look at a series of examples illustrating various formats. Regardless of the format, however, you must always include the *DateAxisRenderer* plug-in in the `<head>` section of the web page.

```
<script type="text/javascript"
    src="../src/plugins/jqplot.dateAxisRenderer.min.js"></script>
```

Handling Date Values in Different Formats

This first example deals with the exchange rate over a period of time, with day-by-day point values. To this end, the input data array should have a sequence of [x, y] values inside it, where x is a date value. The sequence of x values does not comply with the temporal order; jqPlot will sort those points along the x- axis. In Listing 9-27, you use a series of x input values, with different formats for the first five.

Listing 9-27. ch9_13a.html

```
var line1 = [['14-Oct-2012', 1300.41], ['2012-10-15', 1310.50],
             ['2012/10/16', 1322.88], ['17 Oct 2012', 1312.41],
             ['10/18/2012', 1308.16], ['19-Oct-2012', 1310.71],
             ['20-Oct-2012', 1305.01],['21-Oct-2012', 1300.85],
             ['22-Oct-2012', 1290.67]];
```

Next, you have to call the renderer inside the `xaxis` object in `options` in order to activate it. You want to represent the days of the month in which you follow the trend of the exchange values, so you will set the output format without including the year, which remains unchanged. In addition, at the beginning, you want to show the day of the month in numerical form and then the month written with the first three characters, separated by a space. Simply put, in Listing 9-28 the format will be `'%d %b'`, where `%d` stands for day, in digits, and `%b`, for the first three characters of month. The y values are dollars, so you need to add the dollar sign (`$`) as a prefix for the ticks of the y axis. To accomplish this, you must use the `formatString` property for the y ticks as well.

Listing 9-28. ch9_13a.html

```
var options = {
    title: 'Handling Date Values',
    axes:{
       xaxis:{
          renderer: $.jqplot.DateAxisRenderer,
          tickOptions:{
             formatString:'%d %b'
          }
       },
       yaxis:{
          tickOptions:{
             formatString:'$%d'
          }
       }
    }
};
$.jqplot('myChart', [line1], options);
```

Figure 9-26 shows the dollar value, with the prefix `$,` on the y axis and the day and month on the x axis. This is only one of the several formats you can set to represent values on ticks.

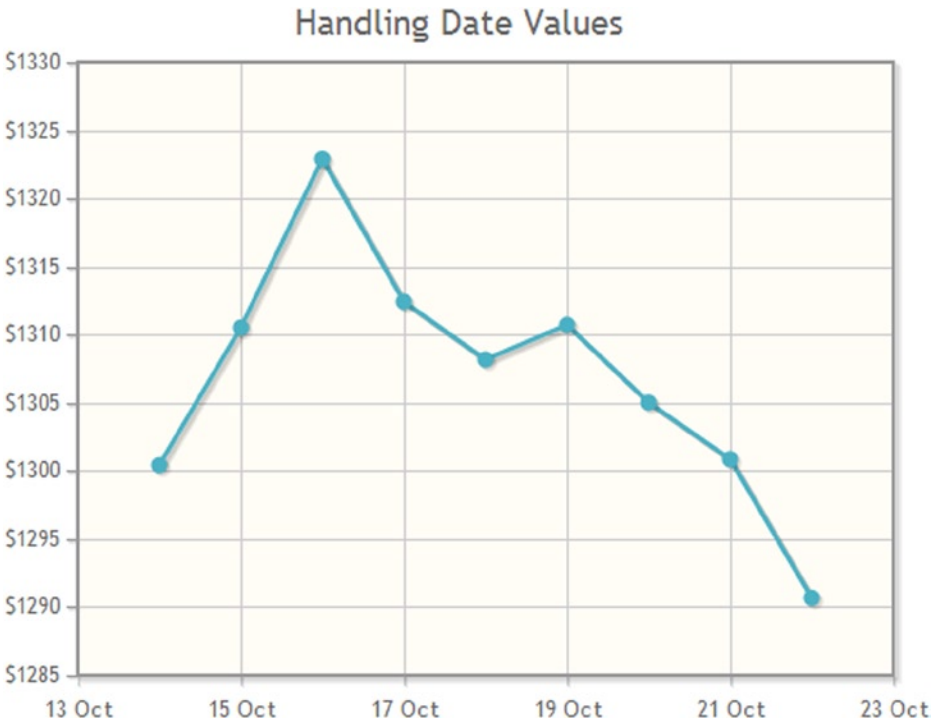


Figure 9-26. *A line chart with date values on the x axis*

Handling Time Values

Let us say you want to draw a chart representing visits to a museum. It is possible to make time explicit in the input data (hours, minutes, seconds). This allows you to handle these time values in the same manner as in previous example (see Listing 9-27), for instance, by creating a chart containing data collected on a given day. Here, too, the date can be set with any of the previously discussed formats. You can express time in various ways: in a 12-hour format, with AM or PM suffixes, or directly, in a 24-hour format, either including or ignoring seconds and minutes. Listing 9-29 illustrates an array with a sequence of time values at 2-hour intervals.

Listing 9-29. ch9_13b.html

```
var line1 = [['2012-10-14 08:00AM', 30],['2012-10-14 10:00AM', 60],
             ['2012-10-14 00:00PM', 120], ['2012-10-14 02:00PM', 60],
             ['2012-10-14 04:00PM', 100], ['2012-10-14 06:00PM', 40]];
```

With regard to the output format, you must remember to manage the time format as well; because you are interested only in the hours of the day, you set `'%R'` as `'formatString'` (see Listing 9-30).

Listing 9-30. ch9_13b.html

```
var options = {
    title: 'Museum Visitors',
    axes:{
       xaxis:{
          label: 'time',
```

```
            renderer:$.jqplot.DateAxisRenderer,
            tickOptions:{
               formatString: '%R'
            }
         },
         yaxis:{
            label: 'visitors'
         }
      }
};
$.jqplot('myChart', [line1], options);
```

The browser will show the chart presented in Figure 9-27.

Figure 9-27. *A bar chart with time values on the x axis*

Highlighting

An eye-catching effect that you can add to your chart is highlighting (i.e., having your plot react to mouseover. For example, the *Highlighter* plug-in will highlight data points near the mouse, with a nice dynamic effect. This can be enhanced by displaying a tool tip with the data point value.

Cursor Highlighter

The following example will serve to familiarize you with highlighting. This functionality is very important and consists in activation of an event when you mouse over particular elements in the chart. Generally, these are elements that represent the data and that, in a line chart, for example, are represented by a point (or, more precisely, by the marker; you will see that this applies to other types of chart as well: a bar in a bar chart, a slice in a pie chart, and so on).

By default the triggered event is just one highlight of the data, represented by a tool tip showing its (x, y) values.

To add these functionalities to your charts, you have to include a set of plug-ins:

```
<script type="text/javascript" src="../src/plugins/jqplot.highlighter.min.js">
</script>
<script type="text/javascript" src="../src/plugins/jqplot.cursor.min.js">
</script>
```

In Listing 9-31, as input data, you use a series of [x, y] pairs with date values on the x axis and numeric values on the y axis.

Listing 9-31. ch9_14a.html

```
var line1 = [['14-Oct-12', 1300.41], ['15-Oct-12', 1310.50],['16-Oct-12', 1322.88],
             ['17-Oct-12', 1312.41],['18-Oct-12', 1308.16],['19-Oct-12', 1310.71],
             ['20-Oct-12', 1305.01],['21-Oct-12', 1300.85],['22-Oct-12', 1290.67]];
```

As you have already seen, in order to handle date values, you need to include the *DateAxisRenderer* plug-in.

```
<script type="text/javascript"
    src="../src/plugins/jqplot.dateAxisRenderer.min.js"></script>
```

In Listing 9-32, you see the `options` object, containing two new objects: `highlighter` and `cursor`.

Listing 9-32. ch9_14a.html

```
var options = {
    title: 'Data Point Highlighting',
    axes:{
       xaxis:{
          renderer: $.jqplot.DateAxisRenderer,
          tickOptions:{
             formatString: '%b %#d'
          }
       },
       yaxis:{
          tickOptions:{
             formatString: '$%d'
          }
       }
    },
    highlighter:{
       show: true,
       sizeAdjust: 7.5
    },
```

```
    cursor:{
       show: false
    }
};
```

In Figure 9-28 a tool tip appears when the cursor moves over a data point on the chart. By default this tool tip reports both x and y values, separated by a comma, using the axis formatters, but this can be customized with a different format string.

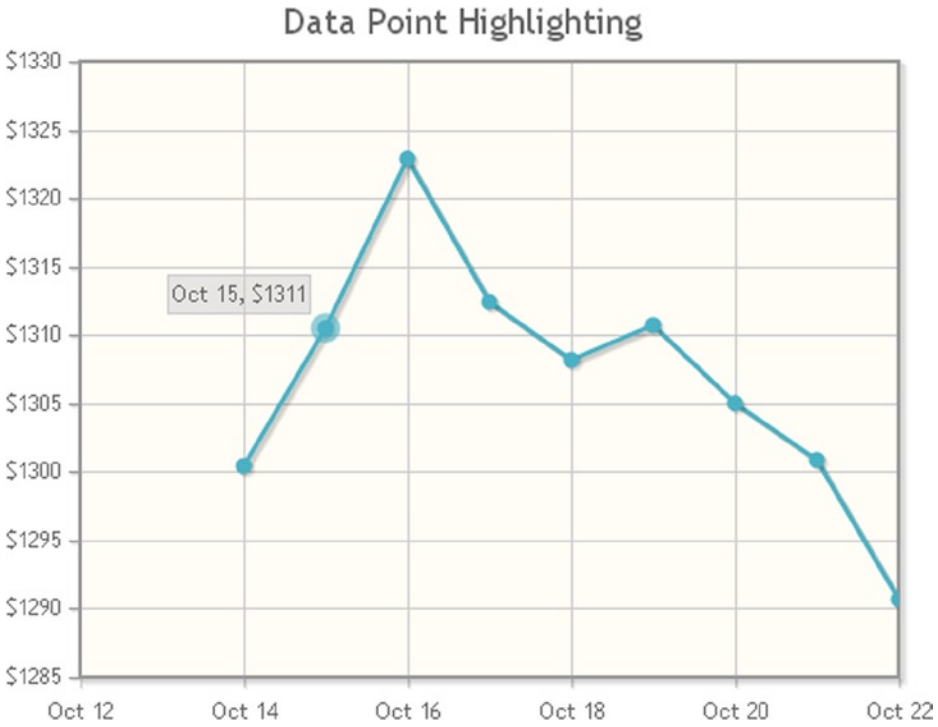


Figure 9-28. *Data point highlighting on a line chart*

In Listing 9-32, you will note that the cursor has been disabled, by setting its `show` property to `'false'` (it is enabled by default). Enabling it, as in Listing 9-33, you will see the mouse cursor changing when it enters the graph area and displaying an optional tool tip in the bottom-right corner, reporting the mouse position. The tool tip can be in a fixed location, or it can follow the mouse. The pointer style, set to `'crosshair'` by default, can also be customized.

Listing 9-33. ch9_14b.html

```
    ...
    highlighter: {
       show: true,
       sizeAdjust: 7.5
    },
```

```
    cursor: {
       show: true,
       tooltipLocation:'ne'
    }
});
```

Figures 9-29 shows a tool tip reporting the cursor coordinates. Note that the cursor is represented by a black cross in the middle of the chart.

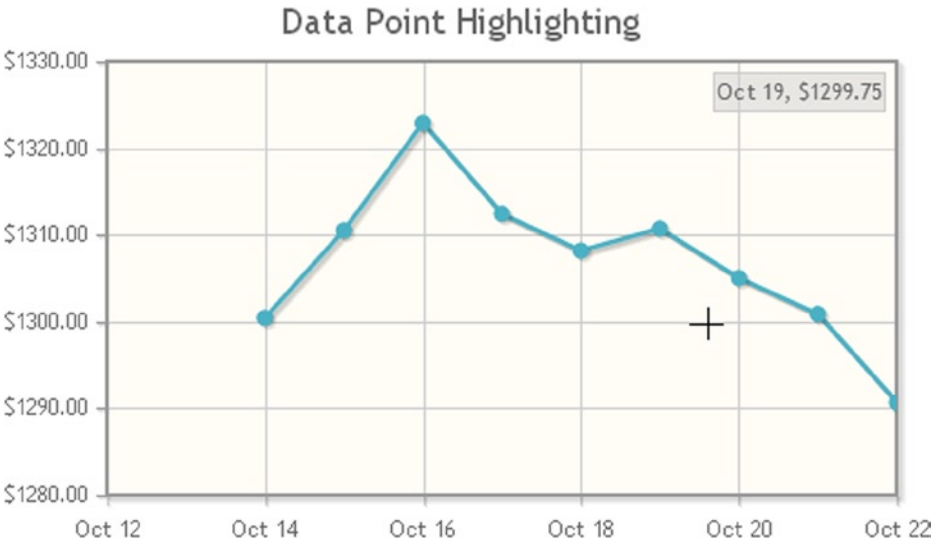


Figure 9-29. *A line chart showing the cursor coordinates*

Highlighting with HTML Format

You can change the content of a tool tip, using HTML tags as format. This makes the possibilities of customization almost unlimited. In fact, you can think of the tool tip as a little web page in which to add any type of element, such as an image or an anchor link (for more details, see Chapter 10). For example, you can use the settings shown in Listing 9-34 with an HTML format string assigned to the `formatString` property.

Listing 9-34. ch9_14c.html

```
highlighter: {
    show: true,
    sizeAdjust: 7.5,
    showMarker: false,
    tooltipAxes: 'xy',
    yvalues: 4,
    formatString:'<table class="jqplot-highlighter"> \
                  <tr><td>date:</td><td>%s</td></tr> \
                  <tr><td>value:</td><td>%s</td></tr></table>'
},
```

As a result, the tool tip with the content will behave like a small HTML page, as shown in Figure 9-30.

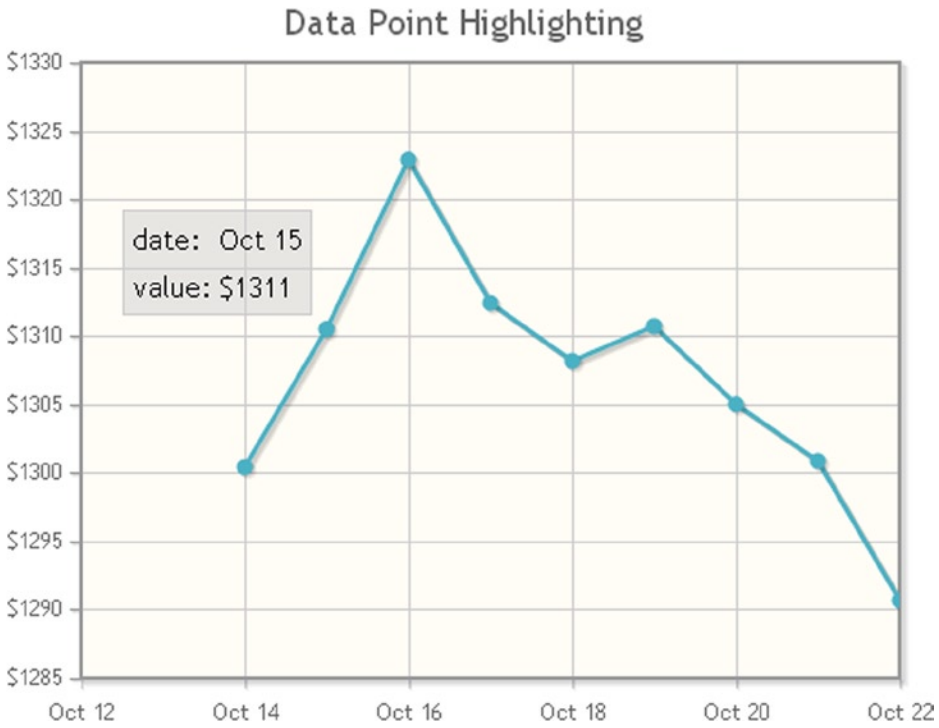


Figure 9-30. *A line chart with an HTML tool tip*

Interacting with the Chart: Limit Lines and Zooming

Once you have a line chart with its graphics and elements well set, the next step is to introduce interactive elements. For instance, it may be necessary for the user to employ threshold values in order to see which data are external to these values. The user mayalso need to vary this threshold to determine which data lie inside and which lie outside it. Often, a lot of data are represented. In this case, the user may need to analyze only a detail.

The jqPlot library provides a solution for both cases with limit lines and zooming. Let us look at some examples addressing these issues in detail.

Drawing a Limit Line on the Chart

Another feature that can be very useful is the *CanvasOverlay* plug-in. It enables you to draw horizontal and vertical lines on your charts, with the purpose of indicating a limit, a threshold, or a deadline or of delimiting a particular range. This can be done by including the *CanvasOverlay* plug-in in the web page:

```
<script type="text/javascript"
    src="../src/plugins/jqplot.canvasOverlay.min.js"></script>
```

By including this plug-in, you have a new object in `options`: `canvasOverlay`. Within this object, you will define an array of objects with their properties. Each of these objects will be represented by a line drawn on the canvas on which jqPlot creates your chart. Five types of objects are already defined in `canvasOverlay`:

- `horizontalLine`
- `verticalLine`
- `dashedHorizontalLine`
- `dashedVerticalLine`
- `Line (generic)`

To see how to insert these limit lines in your charts, let us start from a simple line chart in which you want to show two horizontal limit lines with different colors: a red line marking the upper limit, and a dashed blue line, the lower limit.

In Listing 9-35, you define the two objects: a `horizontalLine` for the lower limit and a `dashedHorizontalLine` for the upper limit. Once you have defined the two lines, you must specify their attributes. The meaning of their attributes, such as y values, `lineWidth`, and `color`, is evident. The `lineCap` property specifies the type of ending placed on the line; it can be `round`, `butt`, or `square`.

Listing 9-35. ch9_15.html

```
$(document).ready(function(){
    var data = [100, 110, 140, 130, 80, 75, 120, 130, 100];
    var options = {
       canvasOverlay: {
          show: true,
          objects: [
          {horizontalLine: {
              y: 70,
              lineWidth: 3,
              color: 'rgb(255, 0, 0)',
              shadow: true,
              lineCap: 'butt'
          }},
          {dashedHorizontalLine: {
              y: 145,
              lineWidth: 4,
              color: 'rgb(0, 0, 255)',
              shadow: false,
              dashPattern: [8, 16],
              lineCap: 'round'
          }}
          ]
       }
    };
    $.jqplot('myChart', [data], options);
});
```

Figure 9-31 displays the two limit lines delimiting the line chart between the values 70 and 145.

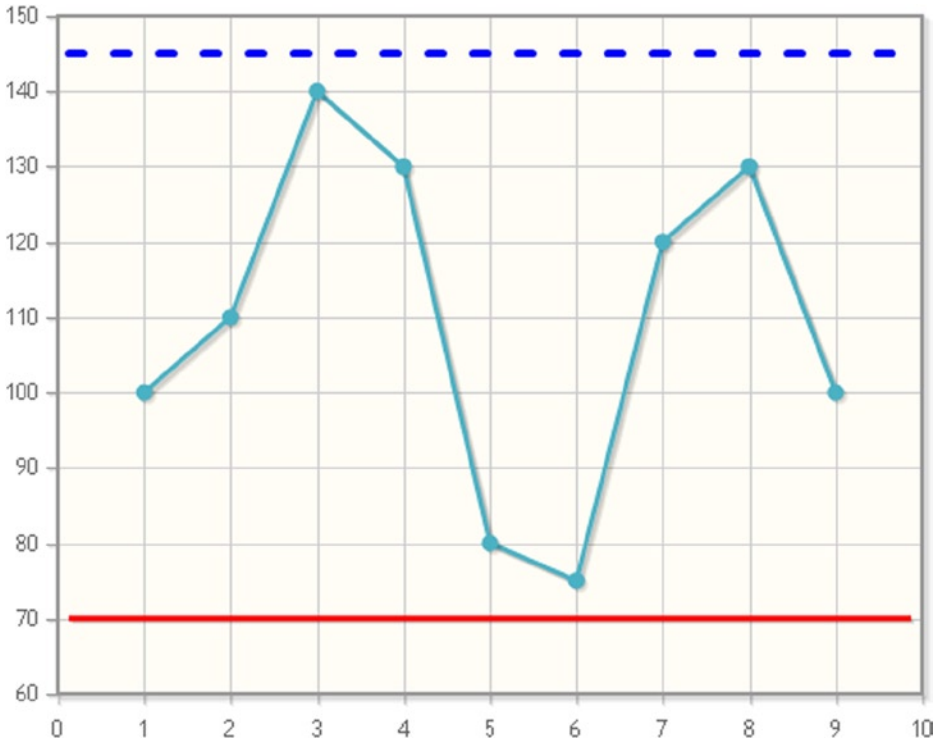


Figure 9-31. *A line chart with lower and upper limits*

You have just seen how to delimit line chart between two limit lines, In more complex cases (but not in this case, which is less common), it may be convenient to vary the value of these limits, and thus be able to move them at will, for instance, by clicking a series of buttons. In the next example, you will continue to implement the current chart by adding buttons that serve to slide the limit lines on the surface of the chart.

Adding Buttons to Your Charts

Using the previous example (see Listing 9-35), you will now see how to add buttons to a chart. Buttons can be placed in any part of a web page, as they are outside the canvas. Here, their function is to allow you to shift the limit lines as you wish, just by clicking them.

For this purpose, you will need four buttons: two to move the limit lines upward and two to move them downward, labeled as follows:

- Low Limit Up
- Low Limit Down
- High Limit Up
- High Limit Down

You can add the four buttons defined in Listing 9-36 anywhere in the `<body>` section of the web page.

Listing 9-36. ch9_16.html

```
<div>
    <button onclick="lineup(myPlot, 'lowlimit')">Low Limit Up</button>
    <button onclick="linedown(myPlot, 'lowlimit')">Low Limit Down</button>
</div>
<div>
    <button onclick="lineup(myPlot, 'hilimit')">High Limit Up</button>
    <button onclick="linedown(myPlot, 'hilimit')">High Limit Down</button>
</div>
```

These rows will generate the four buttons shown in Figure 9-32.

Figure 9-32. *The buttons added to the chart in order to move the limit lines*

In Chapter 2, you were introduced to JQuery User Interface library (jQuery UI) widgets that can be used as controls. Given the potential of this type of control, it is advisable to use the button widget provided by the library (for further information on how to use these widgets, see Chapter 15). If you want to use jQuery UI widgets to replace the four buttons, you need to include the following plug-ins:

```
<link rel="stylesheet" href="http://code.jquery.com/ui/1.10.3/themes/smoothness↩
    /jquery-ui.css" />
<script src="http://code.jquery.com/ui/1.10.3/jquery-ui.min.js"></script>
```

If, however, you would rather refer to the libraries installed locally (see Appendix A), you have to include the following code:

```
<link rel="stylesheet" href="../src/css/smoothness/jquery-ui-1.10.3.custom.min.css" />
<script src="../src/js/jquery-ui-1.10.3.custom.min.js"></script>
```

And, in the <body> section of the HTML page, you add code in Listing 9-37.

Listing 9-37. ch9_16.html

```
<script>
$(function() {
    $('button')
       .button()
       .click(function( event ) {
          event.preventDefault();
       });
});
</script>
```

The buttons are now displayed in jQuery UI style (or, more precisely, with the “smoothness” theme, one of many), as you can see in Figure 9-33.

Figure 9-33. *The same four buttons, but displayed using the jQuery UI*

At this point, these buttons are totally inactive. You need to develop two JavaScript functions; these will be executed when the buttons are pressed. The first function, `lineup()`,will increase the y value of the line passed as argument (high limit or low limit) and then force a new drawing of the chart. The second function, `linedown()`, will decrease the y value. These two functions must be external to the jQuery function `$(document).ready()` (see Listing 9-38).

Listing 9-38. ch9_16.html

```
function lineup(plot, name) {
    var co = plot.plugins.canvasOverlay;
    var line = co.get(name);
    line.options.y += 5;
    co.draw(plot);
}

functionlinedown(plot, name) {
    var co = plot.plugins.canvasOverlay;
    var line = co.get(name);
    line.options.y -= 5;
    co.draw(plot);
}
```

The next step consists in assigning the object returned by the `$.jqplot()`function to a variable:

```
myPlot = $.jqplot('myChart', [data], options);
```

■ **Note** Take care not to write `varmyPlot`, or you will not see any changes in the chart when you press the buttons.

The last step is to name the two lines inside the `canvasOverlay` object, as demonstrated in Listing 9-39.

Listing 9-39. ch9_16.html

```
objects: [
    {horizontalLine: {
       name: 'lowlimit',
       y: 70,
       lineWidth: 3,
       color: 'rgb(255, 0, 0)',
       shadow: true,
       lineCap: 'butt'
    }},
    {dashedHorizontalLine: {
       name: 'hilimit',
       y: 145,
```

```
        lineWidth: 4,
        color: 'rgb(0, 0, 255)',
        shadow: false,
        dashPattern: [8, 16],
        lineCap: 'round'
    }}
]
```

In the end, you get a chart containing the four buttons, as shown in Figure 9-34.

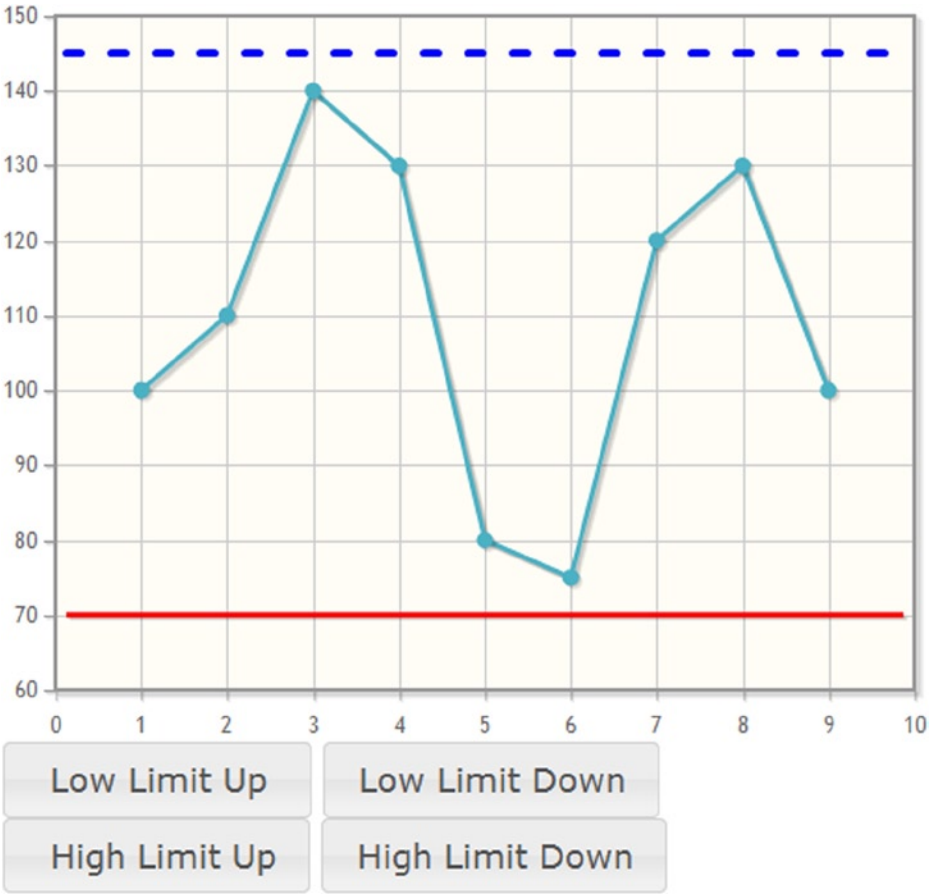


Figure 9-34. *A line chart with a set of buttons that vary the lower and upper thresholds*

Sometimes, you need to add vertical lines to your chart, especially when you have to mark a deadline. In this case we shall work as before, but with some differences. For example, let us say you want to place a single vertical line representing a deadline in your line chart. In this case, you use the code in Listing 9-40.

Listing 9-40. ch9_17.html

```
$(document).ready(function(){
    var data = [100, 110, 140, 130, 80, 75, 120, 130, 100];
    var options = {
       canvasOverlay: {
          show: true,
          objects: [
          {verticalLine: {
             name: 'lowlimit',
             x: 5,
             lineWidth: 3,
             color: 'rgb(50, 200, 50)',
             shadow: true,
             lineCap: 'butt',
             yOffset: 0
          }}
          ]
       }
    };
    myPlot = $.jqplot('myChart', [data], options);
});
```

This time, you will need only two buttons.

```
<div>
    <button onclick="lineright(myPlot, 'lowlimit')">Postpone Deadline</button>
    <button onclick="lineleft(myPlot, 'lowlimit')">Anticipate Deadline</button>
</div>
```

Now, you must develop two JavaScript functions that will shift the limit lines horizontally as the buttons are pressed. Like the JavaScript functions seen previously, the two functions in Listing 9-41 have to be placed external to the jQuery function `$(document).ready()`.

Listing 9-41. ch9_17.html

```
functionlineright(plot, name) {
    var co = plot.plugins.canvasOverlay;
    var line = co.get(name);
    line.options.x += 1;
    co.draw(plot);
}

functionlineleft(plot, name) {
    var co = plot.plugins.canvasOverlay;
    var line = co.get(name);
    line.options.x -= 1;
    co.draw(plot);
}
```

The result is the chart in Figure 9-35, with a green vertical line in the middle. By clicking the two buttons, the line will move to the left if you want to anticipate, or to the right if you want to postpone, the deadline.

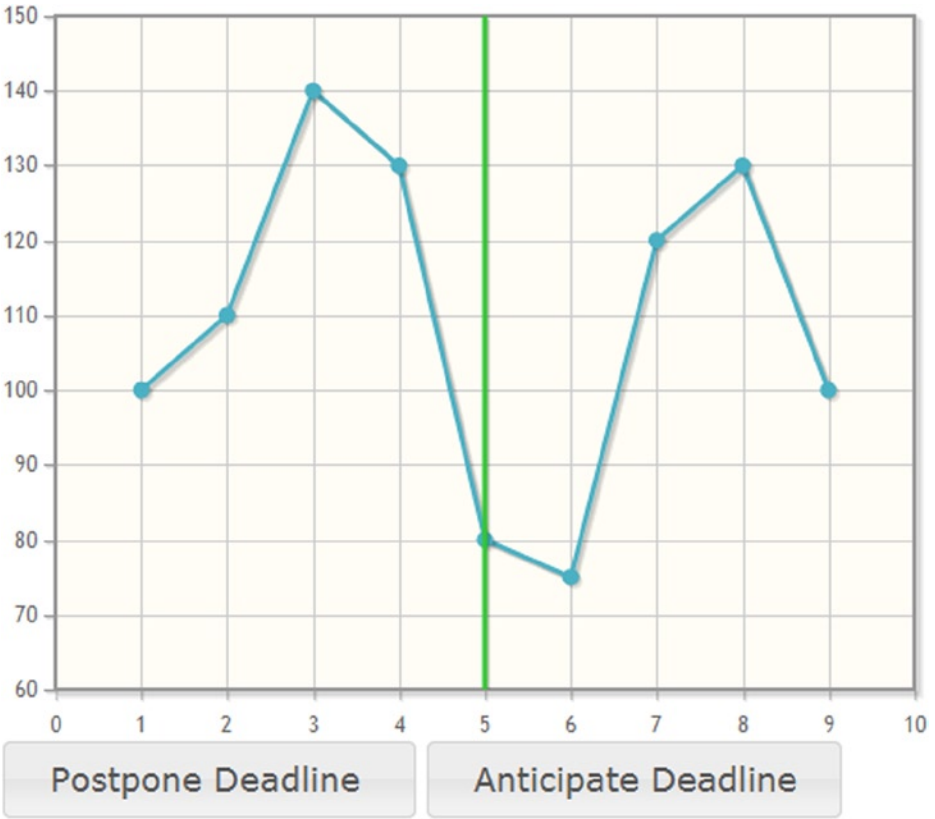


Figure 9-35. *A line chart with a green horizontal limit line*

Zooming

Often, when you are dealing with a large amount of data, you end up with a line made up of thousands of points on your chart. It is precisely in such a case that the zooming function can be indispensable. Starting from a macroscopic view, you can zoom in on part of the line to get a microscopic view of the data.

The *Cursor* plug-in also enables a plot-zooming function. By clicking and dragging the cursor on the plot, you can zoom in on and scroll small sections of your chart. If you double-click, you can reset all and go back to the macroscopic view. Thus, you need to include the *Cursor* plug-in in your web page, and because you have date values on the x axis, the *DateAxisRenderer* plug-in must be included as well:

```
<script type="text/javascript"
    src="../src/plugins/jqplot.dateAxisRenderer.min.js"></script>
<script type="text/javascript" src="../src/plugins/jqplot.cursor.min.js"></script>
```

Or, if you prefer to use a CDN service, you can do so as follows:

```
<script type="text/javascript" src="http://cdn.jsdelivr.net/jqplot/1.0.8/plugins↩
    /jqplot.dateAxisRenderer.min.js"></script>
<script type="text/javascript"
    src="http://cdn.jsdelivr.net/jqplot/1.0.8/plugins/jqplot.cursor.min.js"></script>
```

Listing 9-42 illustrates the availability of a large amount of incoming data.

Listing 9-42. ch9_18.html

```
var data = [["6/22/2012 10:00:00", 110.32], ["6/8/2012 10:00:00", 115.84],
            ["5/26/2012 10:00:00", 121.23], ["5/11/2012 10:00:00", 122.12],
            ["4/27/2012 10:00:00", 120.69], ["4/13/2012 10:00:00",123.24],
            ["3/30/2012 10:00:00", 116.78], ["3/16/2012 10:00:00", 115.16],
            ["3/2/2012 10:00:00", 113.57], ["2/17/2012 10:00:00", 120.45],
            ["2/2/2012 10:00:00", 121.28], ["1/20/2012 10:00:00", 124.7],
            ["1/5/2012 10:00:00", 130.07], ["12/22/2011 10:00:00", 129.36],
            ["12/8/2011 10:00:00", 130.76], ["11/24/2011 10:00:00", 133.96],
            ["11/10/2011 10:00:00", 140.02] ,["10/27/2011 10:00:00", 138.36],
            ["10/13/2011 10:00:00", 140.54], ["9/29/2011 10:00:00", 140.91],
            ["9/15/2011 10:00:00", 140.15], ["9/2/2011 10:00:00", 138.25],
            ["8/25/2011 10:00:00", 137.29], ["8/11/2011 10:00:00", 139.15],
            ["7/28/2011 10:00:00", 144.86], ["7/14/2011 10:00:00", 145.32],
            ["6/30/2011 10:00:00", 148.12], ["6/16/2011 10:00:00", 146.43],
            ["6/2/2011 10:00:00", 147], ["5/19/2011 10:00:00", 144.62],
            ["5/5/2011 10:00:00", 143.2], ["4/21/2011 10:00:00", 144.06],
            ["4/7/2011 10:00:00", 137.45], ["3/24/2011 10:00:00", 138.08],
            ["3/10/2011 10:00:00", 137.92], ["2/25/2011 10:00:00", 131.18],
            ["2/11/2011 10:00:00", 129.64], ["1/28/2011 10:00:00", 133.9],
            ["1/14/2011 10:00:00", 134.25], ["12/31/2010 10:00:00", 137],
            ["12/17/2010 10:00:00", 136.69], ["12/3/2010 10:00:00", 144.87],
            ["11/19/2010 10:00:00", 146.7], ["11/5/2010 10:00:00", 143.97],
            ["10/22/2010 10:00:00", 139.6], ["10/8/2010 10:00:00", 133.39],
            ["9/24/2010 10:00:00", 130.27], ["9/10/2010 10:00:00", 132.75],
            ["8/27/2010 10:00:00", 130.25]];
```

It is very simple to enable the zooming function. Simply set the `zoom` property to `'true'` in `options`, as shown in Listing 9-43.

Listing 9-43. ch9_18.html

```
var options = {
    series: [{
       neighborThreshold: -1
    }],
    axes:{
       xaxis:{
          renderer: $.jqplot.DateAxisRenderer,
          min:'August 1, 2010 16:00:00',
```

```
            tickInterval: '6 months',
            tickOptions: {formatString: '%#m/%#d/%Y'}
        }
    },
    cursor:{
        show: true,
        zoom: true,
        showTooltip: false
    }
};
myPlot = $.jqplot('myChart', [data], options);
```

Or, if you prefer, you can disable the double-clicking that resets the zoom. The *Cursor* plug-in also extends the plot object (the value returned by the `$.jqplot()` function) by using the `resetZoom()` method externally. Moreover, this method can be called from the user code or another HTML element, such as a button, to reset the plot zoom.

You can define this function inside the jQuery `ready()` function:

```
$('.button-reset').click(function() { myPlot.resetZoom() });
```

Then, insert the following row anywhere you want to in order to place the button in the `<body>` section of the web page:

```
<button class="button-reset">Reset Zoom</button>
```

Figure 9-36 offers a sequence of pictures representing the line chart at different moments. The first picture is the line chart as displayed from the browser, without any zooming. The second picture shows an area of the chart selected by the user, with the intention of zooming. The final picture illustrates the result of this zooming. If the user clicks the Reset Zoom button, the browser will display the first picture again.

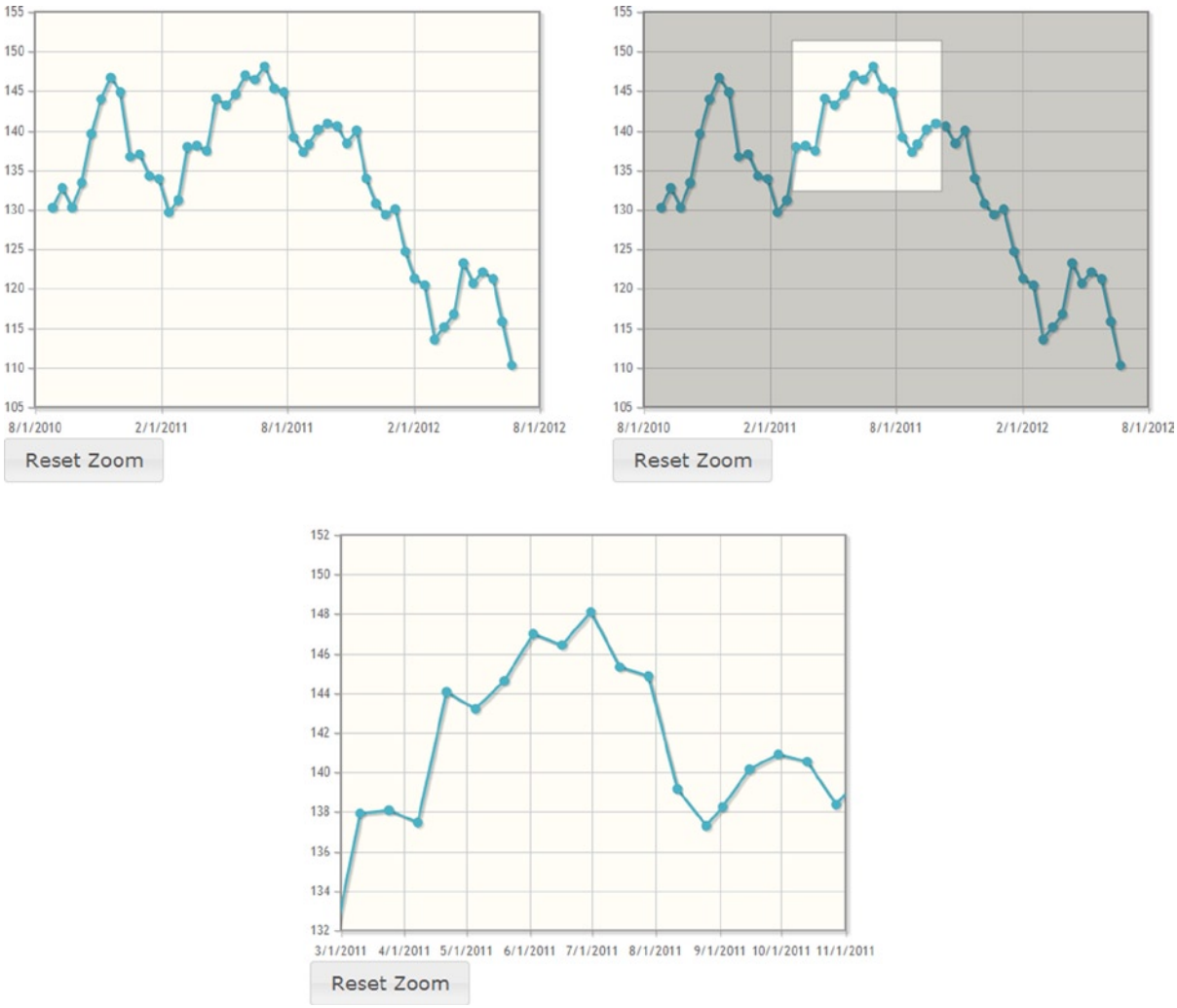


Figure 9-36. *A detail of the line chart extracted by zooming*

Changing Chart Appearance

Thanks to its several plug-ins, jqPlot can render the chart components, including the text, directly on canvas. By now it must be clear that the highlight of the jqPlot library is the potential to change the look of any chart element by varying the default values of the jqPlot properties and of the plug-ins added. But, this is not the only method for effecting such change. If you want to modify the appearance of an element in an HTML page, you have recourse to the CSS style. This is true even for jqPlot elements.

It is possible to refer to several (but not all) jqPlot objects with CSS classes in order to change the style of these objects without having to set their attributes in `options`. The objects can be customized by CSS, using a CSS class such as `.jqplot-*`.

Customizing Text, Using CSS

The jqPlot library provides CSS classes with which you can change some properties without referring to the `options` object. By way of an example, you will use some of these classes to change the text inside a chart. Let us start by implementing a simple multiseries line chart with only a title and an axis label defined in `options` (see listing 9-44).

Listing 9-44. ch9_10a.html

```
$(document).ready(function(){
    var data1 = [1, 2, 3, 2, 3, 4];
    var data2 = [3, 4, 5, 6, 5, 7];
    var data3 = [5, 6, 8, 9, 7, 9];
    var data4 = [7, 8, 9, 11, 10, 11];
    var options = {
       title: 'Multiseries Line Chart,
       axesDefaults: {
          label: 'Axis Label'
       }
    };
    $.jqplot('myChart',[data1, data2, data3, data4], options);
});
```

You add the `<style>` section in Listing 9-45, which can be extracted as a CSS file.

Listing 9-45. ch9_10a.html

```
<style>
.jqplot-title {
    font-family: "Arial Black";
    font-size: 24px;
    color: lightblue;
}

.jqplot-xaxis-label {
    font-size: 24px;
}

.jqplot-axis {
    font-family: "Arial";
    font-size: 16px;
}

.jqplot-xaxis {
    color: green;
}

.jqplot-yaxis {
    color: orange;
    font-weight: bold;
}
</style>
```

Figure 9-37 shows the situation before and after the settings of CSS style in Listing 9-45, allowing us to see the changes made.

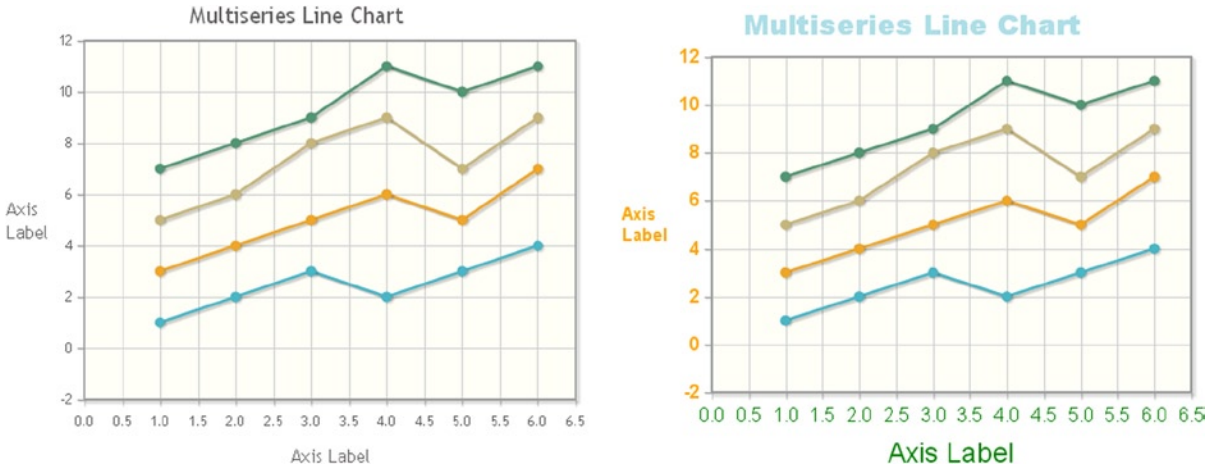


Figure 9-37. *Some CSS styles applied to the tick labels and title*

Changing the Background Color

Continuing with the previous example (see Listing 9-45), you now discover that by simply adding a single property to `options` (highlighted in Listing 9-46), you can obtain a black background. as demonstrated in Figure 9-38.

Listing 9-46. ch9_10b.html

```
var options = {
    title: 'Multiple Data Arrays',
    axesDefaults: {
       label: 'Axis Label'
    },
    grid: {
       background: '#000000'
    }
};
```

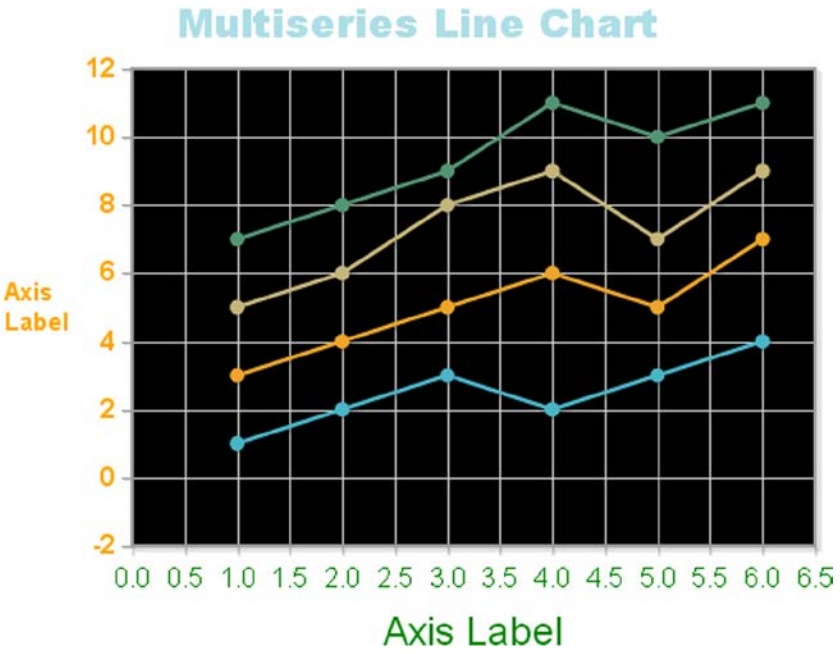


Figure 9-38. *A line chart with a black background*

Further Customization, Using CSS

This time, you will change not only the background to the grid, but also the space that surrounds your chart, making it even more attractive. You will accomplish this by applying CSS styles directly to the chart elements.

For example, counter to the default setting (gray grid, white background), let us say you decide to place your chart on a completely black background. In this case, you need to create a container with a `<div>` element incorporated inside the `myChart` target (another`<div>` element):

```
<div class="chart-container">
    <div id="myChart" style="height:400px; width:500px;"></div>
</div>
```

This container serves to extend the area on which the black background will be placed; we refer to the container by setting its class with `chart-container`.

At this point, the most important thing to keep in mind is that the two elements, the container and the target, can now be suitably characterized by changing their CSS styles. This can be done by specifying attributes for the `.chart-container`, as shown in Listing 9-47 (as for the elements of the target, these have already been set, using the `.jqplot-*` classes). In the `.chart-container` class, you set the `background` property to `'black'`; the size of the container is established with the `width` and `height` properties. You also use the `padding` property in order to better center the target inside the container. The padding clears an area around the content of an element, extending its background color. The four values are, respectively, top, right, bottom, and left padding.

Listing 9-47. ch9_20.html

```
<style type="text/css">
.chart-container {
    background : #000000;
```

```
    padding: 30px 0px 80px 30px;
    width: 560px;
    height: 330px;
}
...
</style>
```

The combined result emerging from the CSS customization of the container and the target is the chart shown in Figure 9-39.

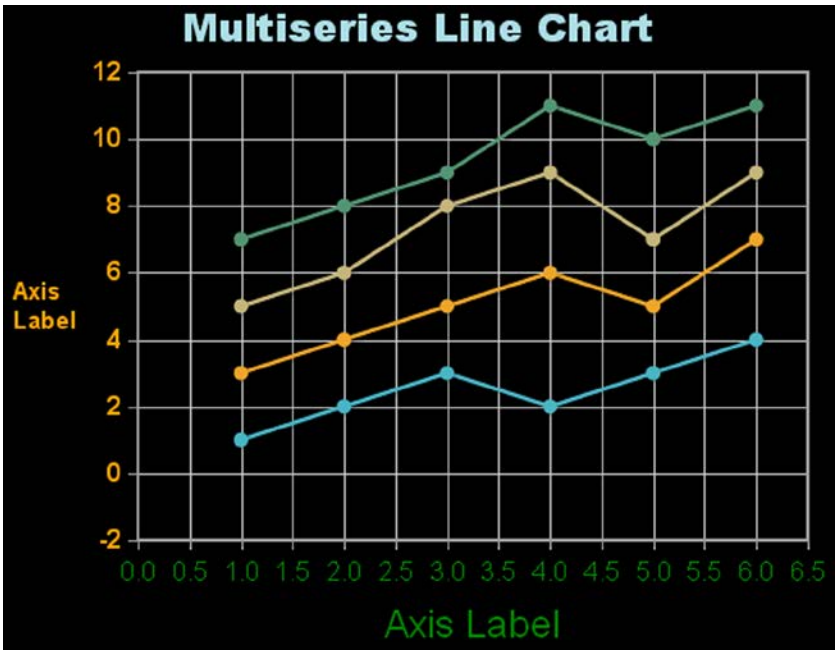


Figure 9-39. *A multiseries line chart with a black background*

Setting the Grid

By default, the grids of your charts are gray. In the previous example (see Listing 9-47), however, you saw how you can change the grid by setting the properties within the `grid` object in `options`. In this example, you continue to modify the same multiseries line chart, but this time, you will focus on the grid properties.

You can change both the grid color and thickness. For instance, you might want a black grid, with increased thickness, in which case you must define the `gridLineColor` and `gridLineWidth` properties. Moreover, sometimes, by default, jqPlot might display your chart with a grid that is too thick and that could hinder rather than aid in readability. In such instances, you will need to reduce the number of ticks. This can be done very easily, by setting the `numberTicks` property in a specific way for each axis within the `axes` object in `options`. Listing 9-48 includes all these changes.

Listing 9-48. ch9_10d.html

```
var options = {
    title: 'Multiseries Line Chart',
    axesDefaults: {
       label: 'AxisLabel'
    },
    grid: {
       background: '#000000',
       gridLineColor: '#ffffff',
       gridLineWidth: 2
    },
    axes: {
       xaxis: {
          numberTicks: 5,
          min: 0,
          max: 8
       },
       yaxis: {
          numberTicks: 3,
          min: 0,
          max: 12
       }
    }
};
```

In the end, you get a new chart with the desired grid (see Figure 9-40).

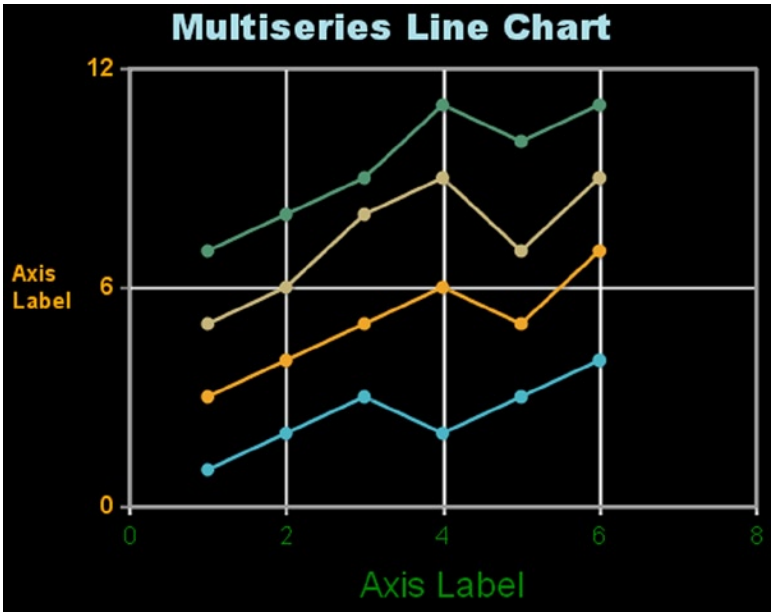


Figure 9-40. *A multiseries line chart with a customized grid*

Note that there is a gray outline delimiting the chart: the border. You can change that as well, or disable it, as in Listing 9-49. You can set the `drawBorder` property to `'false'` and disable the `shadow`, too.

Listing 9-49. ch9_20e.html

```
grid: {
    drawBorder: false,
    shadow: false,
    gridLineColor: '#000000',
    gridLineWidth: 2,
},
```

Following these changes, you obtain a more readable grid, such as the one in Figure 9-41, which has a border the same color as the grid (white).

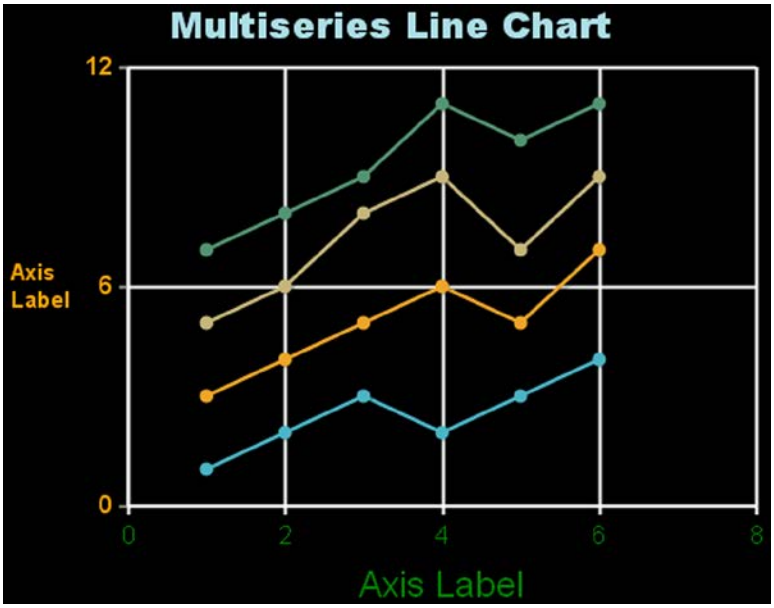


Figure 9-41. *A more readable multiseries line chart, with a customized grid*

Working with Areas on Line Charts

So far, you have seen that the line chart basically consists of sets of points connected by lines, describing trends of a certain size. Now, you may find that such a view is somewhat limited. Often, the most interesting part of a line chart is the area that a line (or several lines) delimits in some way.

Area Charts

A line chart can be converted into an area chart. In this example, you will use the multiseries line chart you have already created (see Listing 9-50), effecting changes in order to get a new chart mixing areas and lines. Here, you will see that very few changes need to be made to achieve the desired effect.

Listing 9-50. ch9_22a.html

```
$(document).ready(function(){
    var data1 = [1, 2, 3, 2, 3, 4];
    var data2 = [3, 4, 5, 6, 5, 7];
    var data3 = [5, 6, 8, 9, 7, 9];
    var data4 = [7, 8, 9, 11, 10, 11];
    var options = {
       title:'Multiple Data Arrays'
    };
    $.jqplot ('myChart', [data1, data2, data3, data4], options);
});
```

First, in `options` you have to insert the `fill` attribute in the series that you want to represent as an area. This time, you choose `seriesDefaults` to apply the representation by area to all series. In this way, you obtain an area chart. To make the chart nicer, you can add other options, such as smoothing (see Listing 9-51).

Listing 9-51. ch9_22a.html

```
var options = {
    title: 'Multiple Data Arrays',
    seriesDefaults: {
       showMarker: false,
       rendererOptions: {
         smooth: true
       },
       fill: true
    }
};
```

But, when running the web page, you immediately find that something is wrong: the area of the last series is covering the others (see Figure 9-42).

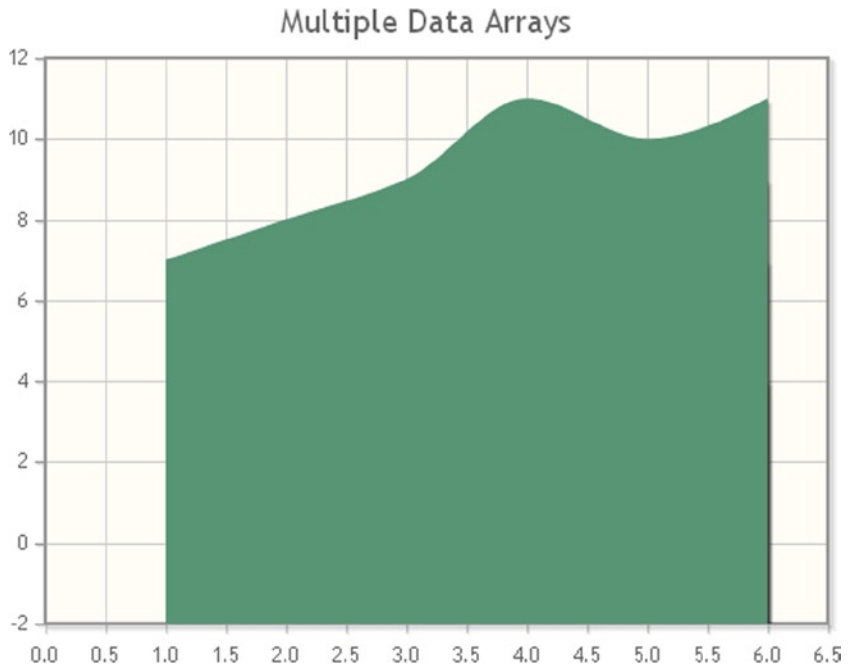


Figure 9-42. *An area chart with one series covering the others*

Before applying the `fill` attribute to series, you need to consider the order in which the corresponding areas of the series should be represented. In this case, it is only necessary to order the sequence of series in a different way:

```
$.jqplot ('myChart', [data4, data3, data2, data1], options);
```

The result is an accurate area chart, as seen in Figure 9-43.

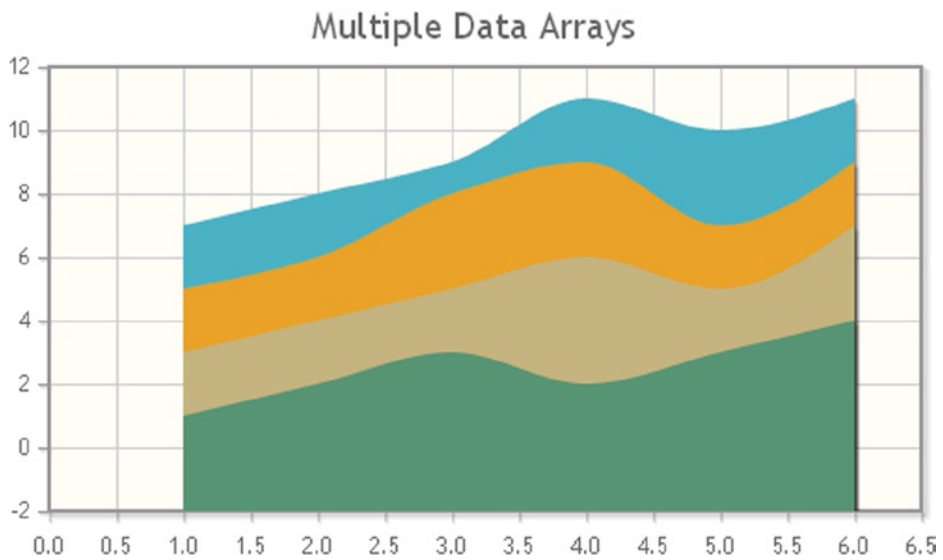


Figure 9-43. *A multiseries area chart rendered correctly*

Line and Area Charts

Mixing lines and areas in the same chart can also create a very nice effect. Continuing with the previous example (see Listing 9-51), instead of setting the `fill` property in `seriesDefaults` and therefore applying the fill for all the series, you can decide to do so series by series in order to choose which series must be represented as area and which as line, as shown in Listing 9-52.

Listing 9-52. ch9_22b.html

```
var options = {
    title:'Multiple Data Arrays',
    seriesDefaults: {
       showMarker: false,
       rendererOptions: {
          smooth: true
       }
    },
    series: [{}, {fill: true}, {}, {fill: true}]
};
```

Figure 9-44 shows how is possible to combine line and area charts.

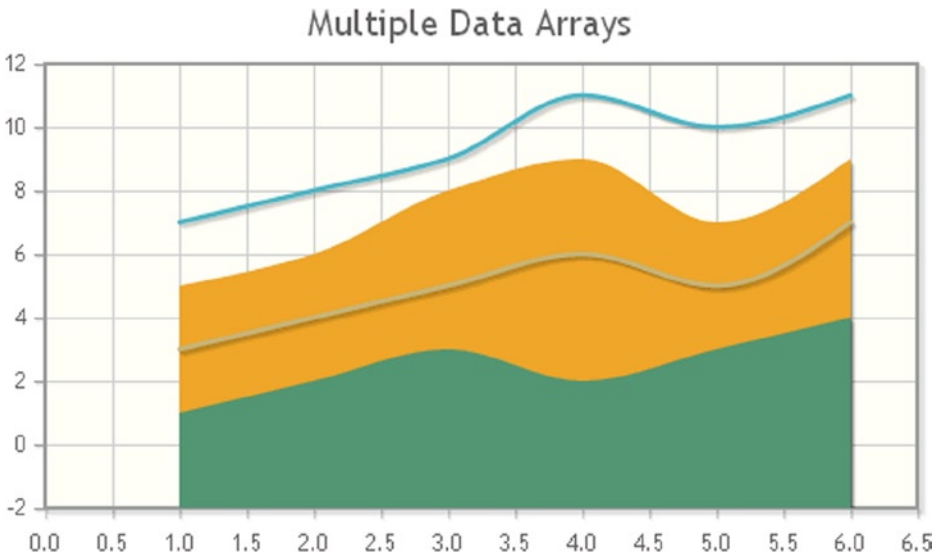


Figure 9-44. *A combined line and area chart*

Band Charts

Band charts (also called high-low line charts or range charts) are a type of chart that combines the features of an area chart with those of a line chart.

A band chart is a line chart enhanced with an underlying shaded area (see Figure 9-45). This area represents the upper and lower boundaries of a range of values on the y axis. This range varies with the x, such that, in the end, you have a band.

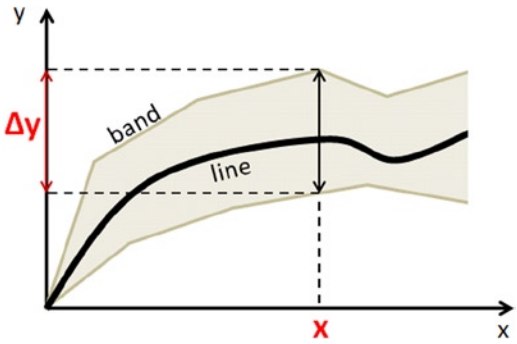


Figure 9-45. *A band chart*

You can use a band to indicate a particular interval on the y axis, which varies with the value on the x axis, correlated with the trend of a line inside it, illustrating, for example, confidence intervals or error bands. Another use might be to highlight a distribution that varies with time and the line showing the arithmetic mean.

Using jqPlot, the bands can be automatically computed or manually assigned. If assigned manually, the bounds of the band must be supplied as two arrays of [x, y] values. The first array delimits the lower bound line; the second array, the upper bound line. These two arrays are joined as the two elements of another array that is passed to the `bandData` property inside `options`.

First, let us define a data array with pairs of [x, y] values and the band array `bdata`, containing the two arrays: lower bound line and upper bound line (see Listing 9-53).

Listing 9-53. ch9_24a.html

```
var data         = [[10,100],[20,110],[30,140],[40,130],
            [50,80],[60,75],[70,120],[80,130],[90,100]];
varbdata =[ [[10,90],[20,100],[30,130],[40,120],
             [50,70],[60,65],[70,110],[80,120],[90,90]],
            [[10,110],[20,120],[30,150],[40,140],
             [50,90],[60,85],[70,130],[80,140],[90,110]]];
```

Then, in `options`, use the code shown in Listing 9-54. Figure 9-46 presents the resulting banded-line chart.

Listing 9-54. ch9_24a.html

```
var options = {
    series: [{
       rendererOptions: { bandData: bdata }
    }],
    seriesDefaults: {
       shadow: false,
       showMarker: false
    }
};
$.jqplot ('myChart', [data], options);
```

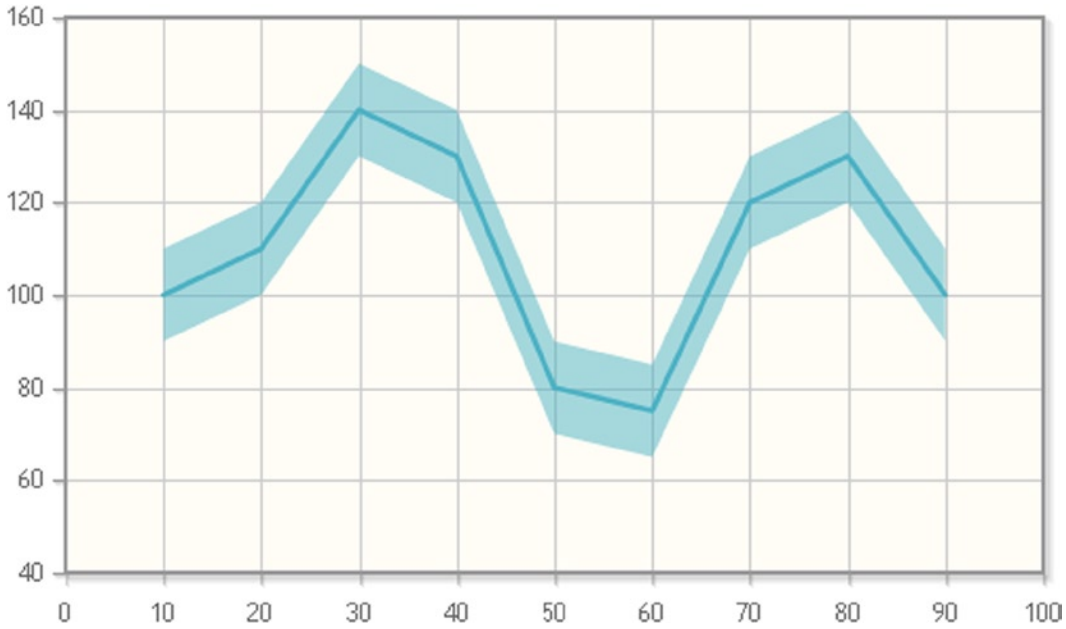


Figure 9-46. *A banded-line chart*

If you choose to draw a smooth-line chart, the band will be smoothed out as well. Listing 9-55 gives the code, and Figure 9-47, the outcome.

Listing 9-55. ch9_24b.html

```
rendererOptions: {
    bandData: bdata,
    smooth: true
}
```

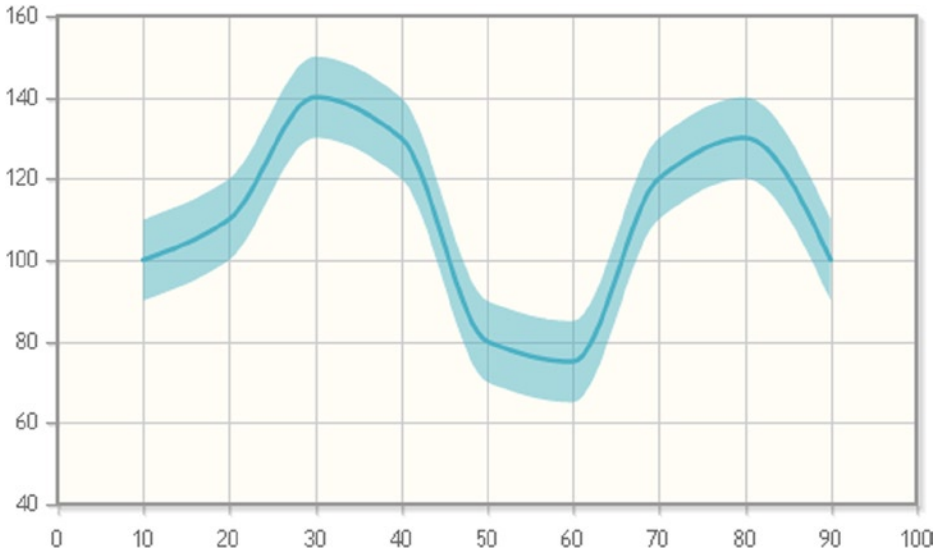


Figure 9-47. *A smooth-banded-line chart*

The number of points in the band data arrays does not have to correspond to the number of points in the data series. Also, band data will be drawn as smoothed lines if the data series is smoothed. The band does not have to be symmetrical, with respect to the main line. The band can be made asymmetrical by inserting an array with asymmetrical y values in the `bdata` array, as demonstrated in Listing 9-56.

Listing 9-56. ch9_24c.html

```
varbdata =[ [[10,90],[30,100],[40,100],[50,70],
             [60,65], [70,110],[80,120],[90,90]],
            [[10,110],[30,150],[40,140],[50,120],
             [60,85], [70,130],[80,140],[90,110]] ];
```

Now, the band in Figure 9-48 is not symmetrical, with respect to the main line.

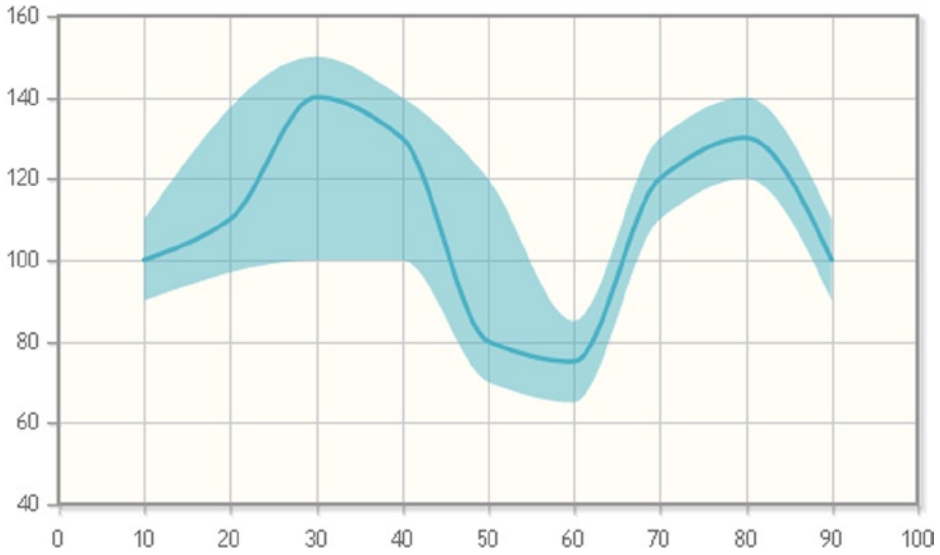


Figure 9-48. *A nonuniform banded-line chart*

But, providing band data is not mandatory; they can be automatically computed by jqPlot. To activate this feature without using any arrays, you have to set the `bands` object's `show` property to `'true'` in `rendererOptions`, as in Listing 9-57. As you can see in Figure 9-49, by default the band interval covers +/-3 percent of the y value of the main line.

Listing 9-57. ch9_24d.html

```
series: [{
    rendererOptions: {
       bands: { show: true},
       smooth: true
    }
}],
```

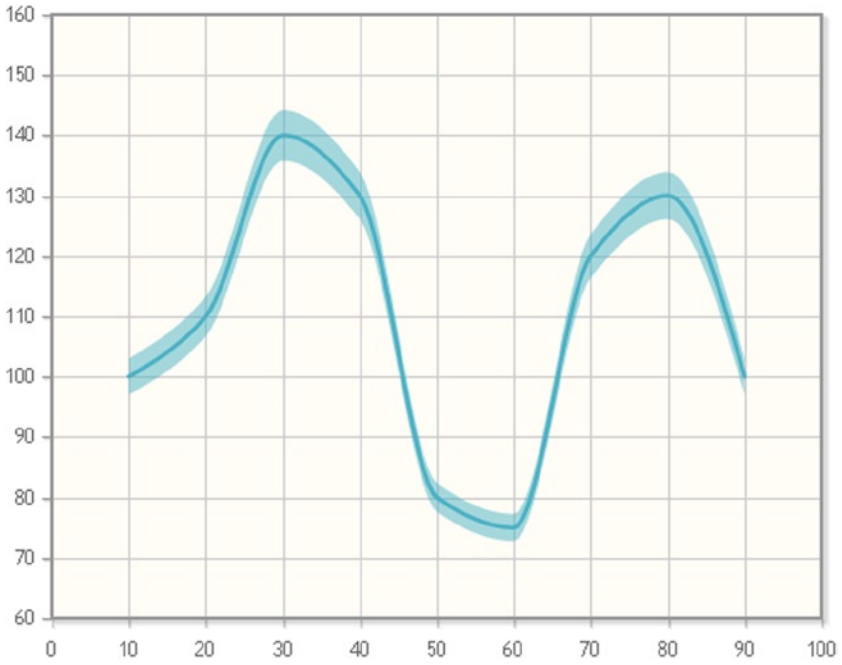


Figure 9-49. *A banded-line chart with a band interval of +/-3 percent*

Filling Between Lines in a Line Chart

You have just learned about bands. Why not fill the area between two series lines? Even this task is possible with jqPlot, as well. By setting the properties within the `fillBetween` object, it is possible to control the area between two lines on a plot.

Here, you are starting with a very simple multiseries line chart (the same example used for other cases), described in Listing 9-58.

Listing 9-58. ch9_5a.html

```
$(document).ready(function(){
    var data1 = [1, 2, 3, 2, 3, 4];
    var data2 = [3, 4, 5, 6, 5, 7];
    var data3 = [5, 6, 8, 9, 7, 9];
    var data4 = [7, 8, 9, 11, 10, 11];
    var options = {
       title:'Multiple Data Arrays',
    };
    $.jqplot('myChart', [data1, data2, data3, data4], options);
});
```

Using this multiseries line chart, you can consider every series, with an index starting from 0, for the first series; 1, for the second series; 2, for the third; and so on.

So, if you want to fill the area between two lines, you need to specify in the `series1` and `series2` attributes the two indexes corresponding to them. For instance, if you want to fill the area between the second and the fourth series, you have to set `series1` to 1 (second series) and `series 2` to 3 (fourth series), as shown in Listing 9-59. Optionally, you can set the color of the delimited area by using the `color` attribute or, better, with an `rgba()` function.

Listing 9-59. ch9_25.html

```
var options = {
    title: 'Multiple Data Arrays',
    fillBetween: {
       series1: 1,   //second series
       series2: 3,   //fourth series
       color: "rgba(10, 120, 130, 0.7)"
    }
});
```

In Figure 9-50, you can see that the selected area, between the second and the fourth series, is colored.

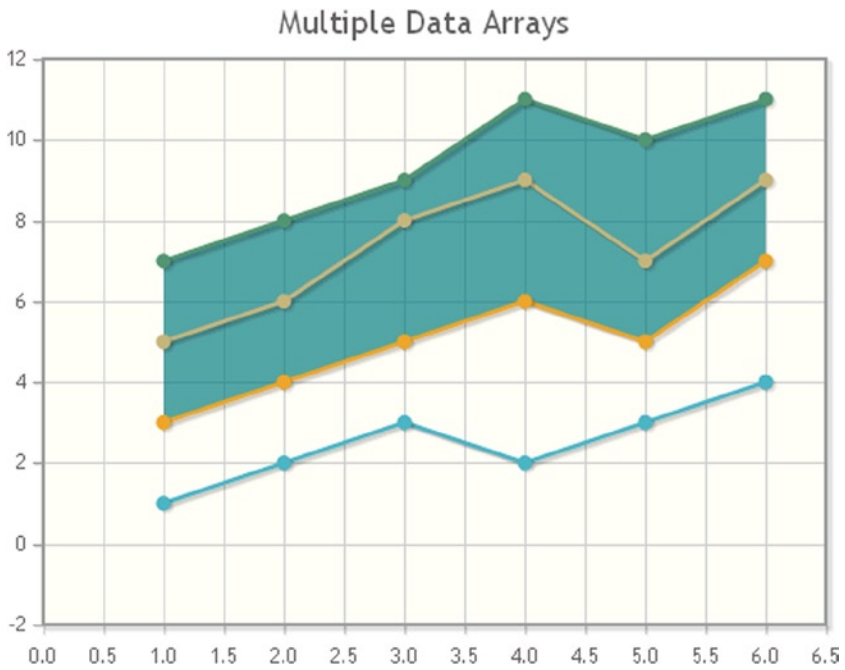


Figure 9-50. *A multiseries line chart with a colored area between two lines*

You can bind a JavaScript function to a button that serves to update the plot's settings for each series and then replot everything. To do this, let us add the function in Listing 9-60 inside the jQuery `ready()` function.

Listing 9-60. ch9_26.html

```
$("button[name=changeFill]").click(function(e) {
    plot1.fillBetween.series1 = parseInt($("input[name=series1]").val());
    plot1.fillBetween.series2 = parseInt($("input[name=series2]").val());
    plot1.replot();
});
```

For the previous JavaScript function to work, you need to assign the variable `plot1` with the value returned by the function `$.jqplot()`:

```
plot1 = $.jqplot ('myChart', [data1, data2, data3, data4], options);
```

And, in the `<body>` section of the web page, you must add two input text areas and a button, as shown in Listing 9-61.

Listing 9-61. ch9_26.html

```
<label for="series1">First Series: </label>
<input type="text" name="series1" value="1" />
<label for="series2"> Second Series: </label>
<input type="text" name="series2" value="3" />
<button name="changeFill">Change Fill</button>
```

The result is given in Figure 9-51.

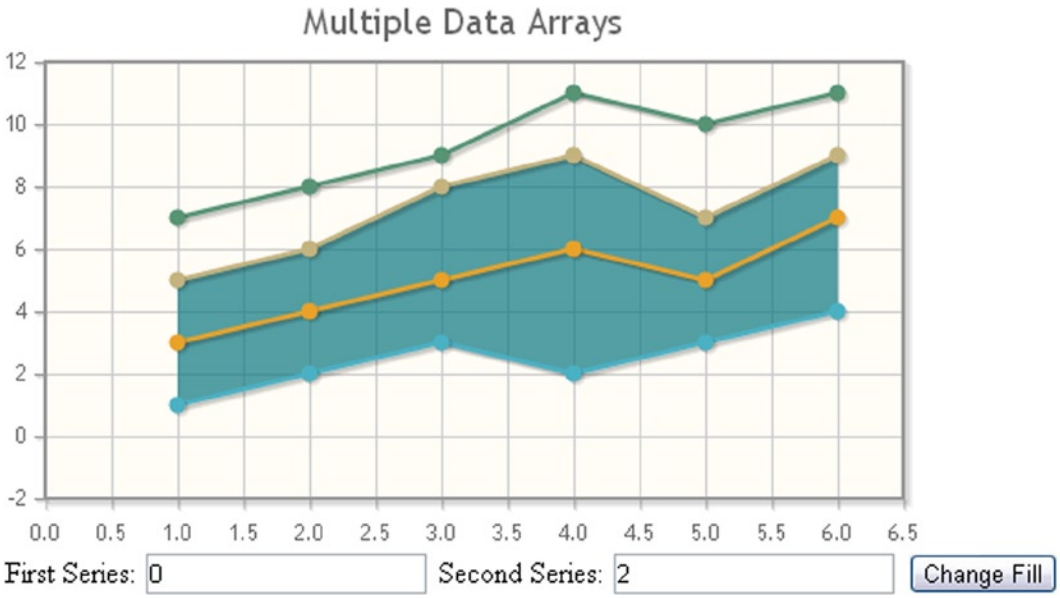


Figure 9-51. *A multiseries line chart with a selectable colored area*

To replace the simple HTML controls with a jQuery UI widget, you have to make some changes to the code in order to integrate it, as reported in Listing 9-62.

Listing 9-62. ch9_26ui.html

```
<link rel="stylesheet" href="../src/css/smoothness/jquery-ui-1.10.3.custom.min.css" />
<script src="../src/js/jquery-ui-1.10.3.custom.min.js"></script>
...
$("button[name=changeFill]").click(function(e) {
    plot1.fillBetween.series1 = parseInt($("#combobox").val());
```

```
    plot1.fillBetween.series2 = parseInt($("#combobox2").val());
    plot1.replot();
});
...
<div class="ui-widget">
<label>First Series : </label>
<select id="combobox">
    <option value="0">1</option>
    <option value="1">2</option>
    <option value="2">3</option>
    <option value="3">4</option>
</select>
</div>
<div class="ui-widget">
<label>Second Series : </label>
<select id="combobox2">
    <option value="0">1</option>
    <option value="1">2</option>
    <option value="2">3</option>
    <option value="3">4</option>
</select>
</div>
<button name="changeFill">Change Fill</button>
<script>
$(function() {
    $( 'button')
       .button()
       .click(function( event ) {
          event.preventDefault();
      });
});
</script>
```

The result is the chart shown in Figure 9-52.

Figure 9-52. *A multiseries line chart with a selectable colored area*

Trend Lines

jqPlot is really full of surprises. In addition to all the things you have already seen regarding the line chart, jqPlot can calculate and represent trend lines. These are generally straight lines drawn in a chart, but sometimes they can be exponential (if they are linear in a log scale). A trend line indicates the general pattern or direction of the series data plotted in a chart. The line is drawn by using statistical techniques. This function is performed by another plug-in: *Trendline*.

To enable this function, you need to include the plug-in in the web page:

```
<script type="text/javascript" src="../src/plugins/jqplot.trendline.min.js"></script>
```

After that, you need only activate the plug-in, adding the row that enables it, as in Listing 9-63.

Listing 9-63. ch9_27a.html

```
$(document).ready(function(){
    var data = [100, 110, 140, 130, 135, 132, 140, 135, 142]
    $.jqplot.config.enablePlugins = true;
    $.jqplot ('myChart', [data]);
});
```

With these few lines it is possible to obtain a trend line, as in Figure 9-53.

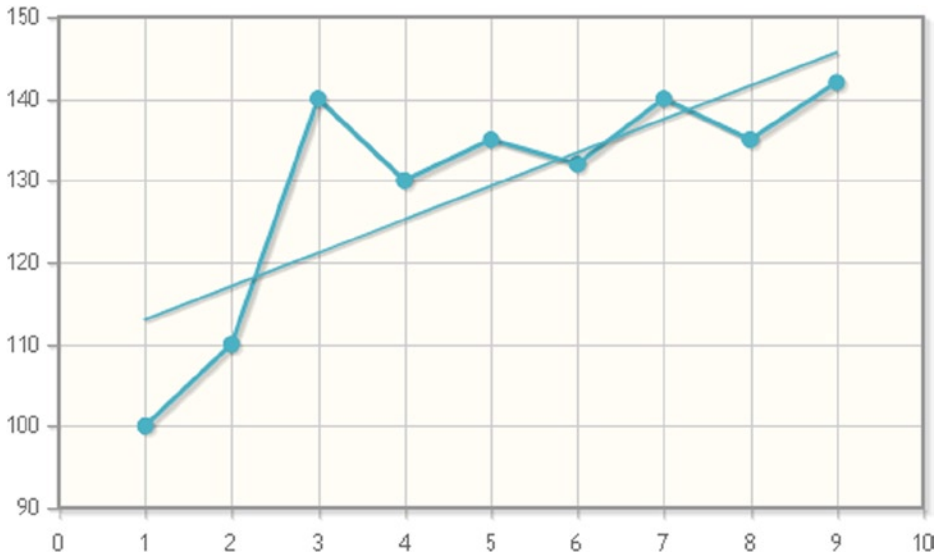


Figure 9-53. *The linear trend line of a line chart*

But, if you prefer to express the properties explicitly, you can do so through the use of options. This enables you to handle the trend line in the same way as other objects in the chart. Let us say you would like to change the line's color and increase its thickness to make it stand out (see Listing 9-64).

Listing 9-64. ch9_27b.html

```
var options = {
    seriesDefaults: {
       trendline: {
          show:true,
          color: '#ff0000',
          lineWidth: 4
       }
    }
}
$.jqplot ('myChart', [data], options);
```

You now gain more control of the trend line. Figure 9-54 shows a line with property settings in the `trendline` object (the trend line is thicker and is displayed with a deep red color on the browser).

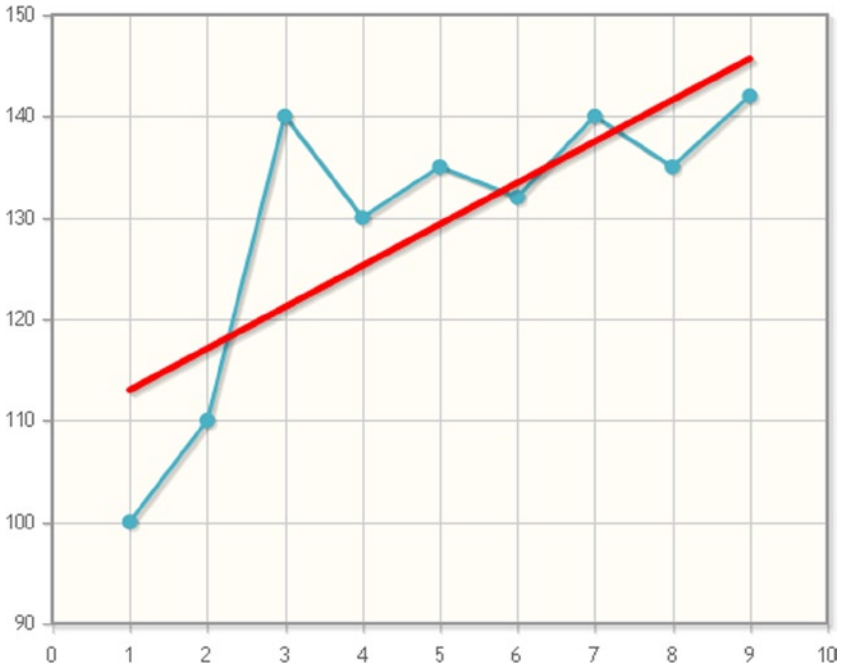


Figure 9-54. *A customized linear trend line in a line chart*

As mentioned earlier, it is possible to use trend line curves, which indicate an exponential trend followed by the points of the chart. Let examine this in the next example, in Listing 9-65.

Listing 9-65. ch9_28.html

```
$(document).ready(function(){
    var data = [[10, 1.44], [30, 6.98], [50, 10.7], [70, 37.5], [90, 78.1]];
    var options = {
       seriesDefaults: {
          trendline: {
             show:true,
             color: '#ff0000',
             lineWidth: 4,
             type: 'exponential'
          }
       }
    }
    $.jqplot ('myChart', [data], options);
});
```

Figure 9-55 presents a line chart in which you have plotted a series of points following an exponential trend. You can, therefore, highlight this with an exponential trend line.

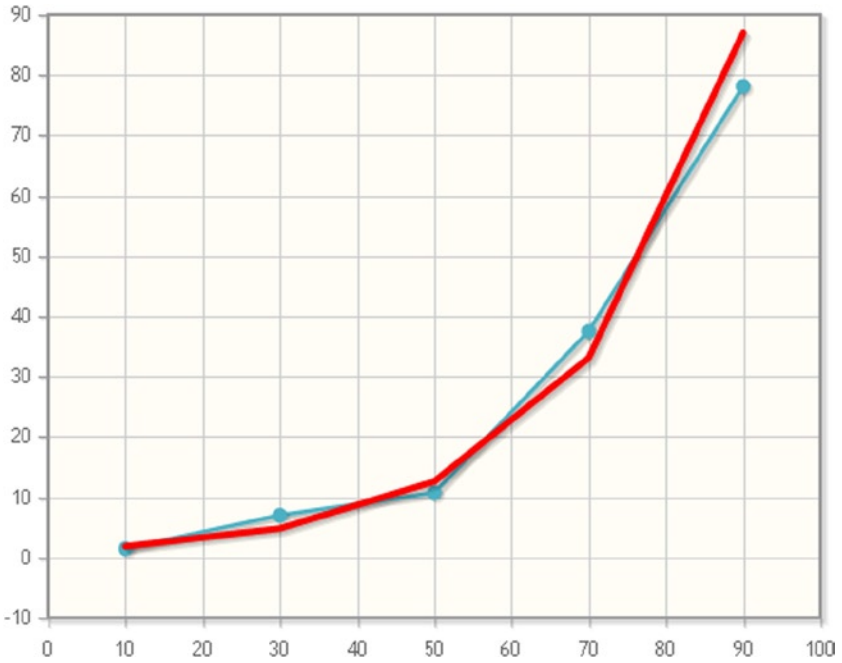


Figure 9-55. *A customized exponential trend in a line chart*

Summary

In this rich chapter, you have become well versed in the jqPlot world. You looked at the many possibilities that this library provides, enabling you to implement line charts to the best of your ability. You learned how to manipulate the basic elements on which to plot your chart, such as **axes** and **ticks**. In particular, you saw how to manage multiple data series in the same chart (**multiseries charts**), adding various graphic effects. You also explored the way in which the jqPlot library allows you to manipulate different formats of **date and time values**. Moreover, you saw how it is possible to customize some elements, using the **HTML format**, along with the **highlighting** of data points. In the final part of the chapter, you dealt with more complex cases, such as generating a **trend line** and working with **band charts**.

In the next chapter, one that is full of arguments, you will face other new concepts, applied this time to bar charts.

CHAPTER 10

Bar Charts with jqPlot

In this chapter, you will deal with another large class of charts: bar charts. In the previous chapter, you were shown ways to characterize line charts, the default chart type in jqPlot. Now, using the *BarRenderer* plug-in, you will discover how the structure of the main jqPlot object is gradually enriched with new properties and objects. Through practical examples, you will see how to change the values of property and object attributes with `rendererOptions`.

Sometimes, it is possible to obtain different representations using the same set of data. Learning how to choose which representation is most suitable to your needs is one of the fundamental objectives of this book. To this end, using one set of data, you will see how to switch from a grouped bar chart to a stacked bar chart, in both cases choosing between a vertical and a horizontal representation.

In addition, you will learn how it is possible to represent a combined chart with the jqPlot library, for example, how to represent a line chart and a bar chart at the same time, and how, simply by slowing down the speed of drawing, you can get a simple but eye-catching animation., You will also become acquainted with a particular type of bar chart, the Marimekko chart, which is implemented by the jqPlot library in a very satisfactory way.

In the last part of the chapter, I will introduce the use of events in jqPlot. This is a complex subject, but thanks to special jQuery functions, you can achieve significant interactive effects with only a few lines of code. The chapter concludes with a typical example in this regard: how to customize tool tips.

Using the *BarRenderer* Plug-In to Create Bar Charts

When you have a set of data that is divided into various categories, and there is a need to compare these categories with one another, then a bar chart may be the representation that is best suited to your needs. You have seen that without including any plug-ins, by default the incoming data are interpreted as points joined to form a line. To tell jqPlot that the incoming data must be used to draw a bar chart, you have to place a set of plug-ins in the `<head>` section of the HTML page:

```
<script type="text/javascript" src="../src/plugins/jqplot.dateAxisRenderer.min.js"></script>
<script type="text/javascript"
    src="../src/plugins/jqplot.canvasAxisTickRenderer.min.js"></script>
<script type="text/javascript" src="../src/plugins/jqplot.categoryAxisRenderer.min.js"></script>
<script type="text/javascript" src="../src/plugins/jqplot.barRenderer.min.js"></script>
```

Or, if you prefer to use a content delivery network (CDN) service, you may do so as follows:

```
<script type="text/javascript"
src="http://cdn.jsdelivr.net/jqplot/1.0.8/plugins/jqplot.dateAxisRenderer.min.js"></script>
<script type="text/javascript"
src="http://cdn.jsdelivr.net/jqplot/1.0.8/plugins/jqplot.canvasAxisTickRenderer.min.js"></script>
```

```
<script type="text/javascript"
src="http://cdn.jsdelivr.net/jqplot/1.0.8/plugins/jqplot.categoryAxisRenderer.min.js"></script>
<script type="text/javascript"
src="http://cdn.jsdelivr.net/jqplot/1.0.8/plugins/jqplot.barRenderer.min.js"></script>
```

To set the input data so that it may be used in a bar chart, you must insert an array in the format [label, y], where x is no longer present, but an indicative label takes its place. This label is often a string value. In fact, when we talk about bar charts, we are no longer interested in following the trend of a variable (y values) in relation to another variable (x values), but rather, are interested in comparing categories, or groups, of data (labels). For this example, you will use five groups, each of which represents a state, reported as a label:

```
var data = [['Germany', 12], ['Italy', 8], ['Spain', 6], ['France', 10], ['UK', 7]];
```

Once you have included the *BarRenderer* plug-in, you have to activate it, assigning its reference to the `renderer` property within the `series` object (see Listing 10-1). You will do the same thing with the second plug-in, *CategoryAxisRenderer*, specifying it only for the `xaxis` object.

Listing 10-1. ch10_01a.html

```
var options = {
    title: 'Foreign Customers',
    series:[{renderer:$.jqplot.BarRenderer}],
    axes: {
       xaxis: {
          renderer: $.jqplot.CategoryAxisRenderer
       }
    }
};
$.jqplot ('myChart', [data], options);
```

Next, in the `<body>` section of the HTML page, you add the following row:

```
<div id="myChart" style="height:300px; width:500px;"></div>
```

In this way, you get a simple bar chart, as shown in Figure 10-1. Every state contained in the data array is represented by a blue bar, with the height corresponding to the y value.

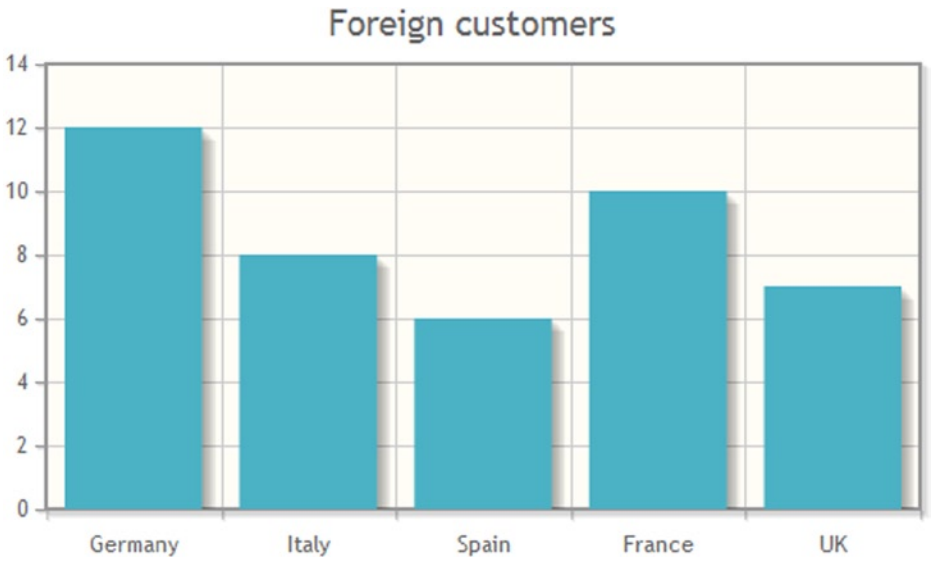


Figure 10-1. *A simple bar chart*

Rotate Axis Tick Labels

Often, it may be necessary or desirable to rotate the tick labels reported on the x axis. For instance, the text may be too long to be reported, and in order to maintain the readability of the labels, you need to write them so that they incline at a certain angle. This rotation is achieved by including the *CanvasTextRenderer* plug-in:

```
<script type="text/javascript" src="../src/plugins/jqplot.canvasTextRenderer.min.js"></script>
```

Or, if you prefer to use a CDN service, you may do so as follows:

```
<script type="text/javascript" src="http://cdn.jsdelivr.net/jqplot/1.0.8/plugins↩
    /jqplot.canvasTextRenderer.min.js"></script>
```

Here, too, simply including the plug-in will not be enough; you must also activate it by passing its reference to the `tickRenderer` property, as demonstrated in Listing 10-2. Then, you need to specify certain properties in `tickOptions`: you set the `angle` property to –30 degrees. With this value, you are indicating the degrees of inclination of the text, with respect to the x axis. If the value is positive, the text will rotate in a clockwise direction; if negative (as in Listing 10-2), counterclockwise.

Listing 10-2. ch10_01b.html

```
var options = {
    title: 'Foreign customers',
    series:[{ renderer: $.jqplot.BarRenderer }],
    axes: {
       xaxis: {
          renderer: $.jqplot.CategoryAxisRenderer,
          tickRenderer: $.jqplot.CanvasAxisTickRenderer,
```

```
            tickOptions: {
               angle: -30,
               fontSize: '10pt'
            }
            }
    }
};
$.jqplot ('myChart', [data], options);
```

If you now load the web page in your browser, at the bottom of your chart,, you will see that all the labels are rotated counterclockwise, with respect to the x axis (see Figure 10-2).

Figure 10-2. *A bar chart with rotated labels on the x axis*

Modify the Space Between the Bars

Working with bar charts, perhaps the most common requirement is to vary the amount of space between bars. This space can be adjusted directly, by setting the `barMargin` property with different values. Because this property does not belong to the `jqplot` object, but is specific to the *BarRenderer* plug-in, you have to specify it within `rendererOptions`. Whenever you include a renderer plug-in, you also include a whole new set of properties not belonging to the original `jqplot` object. So, if you want a value that is different from the default of one of these properties, you will need to write this property in `rendererOptions`, setting the new value. For instance, let us apply a space of 30 pixels between bars, as illustrated in Listing 10-3.

Listing 10-3. ch10_02.html

```
var options = {
    title: 'Foreign Customers',
    seriesDefaults:{
       renderer:$.jqplot.BarRenderer,
       rendererOptions: {
          barMargin: 30
       },
    },
    axes: {
```

Because the width of the chart remains the same, as a consequence of increasing the `barMargin` property to 30, all bars are narrower than before (see Figure 10-3).

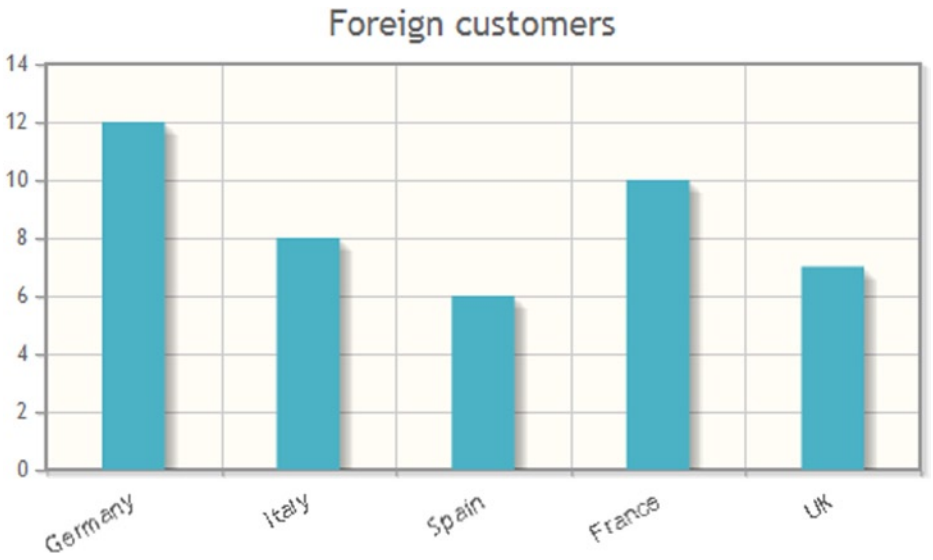


Figure 10-3. *The space between bars is adjustable with a new property introduced by the plug-in*

Adding Values at the Top of Bars

The jqPlot library allows you to handle even point labels. Although you can use them in line charts, as well, point labels are an important component of bar charts. Point labels, if activated, explicitly show the y value above the bars, enhancing the readability of values, especially for the stacked bar chart. To activate this functionality, you need to include another plug-in:

```
<script type="text/javascript" src="../src/plugins/jqplot.pointLabels.min.js"></script>
```

Or, if you prefer to use a CDN service, you may do so as follows:

```
<script type="text/javascript"
    src="http://cdn.jsdelivr.net/jqplot/1.0.8/plugins/jqplot.pointLabels.min.js"></script>
```

You may have noticed that *PointLabels* is not a renderer plug-in and that it is therefore already active. This time, there will be no need to pass a reference in the `renderer` property. The process is very simple and quick: in `options`, you set the `show` property of the `pointLabels` object to `'true'` (see Listing 10-4).

Listing 10-4. ch10_03.html

```
seriesDefaults:{
    renderer:$.jqplot.BarRenderer,
    pointLabels: { show: true }
},
```

As shown in Figure 10-4, with the point labels activated, the y value will appear above each bar.

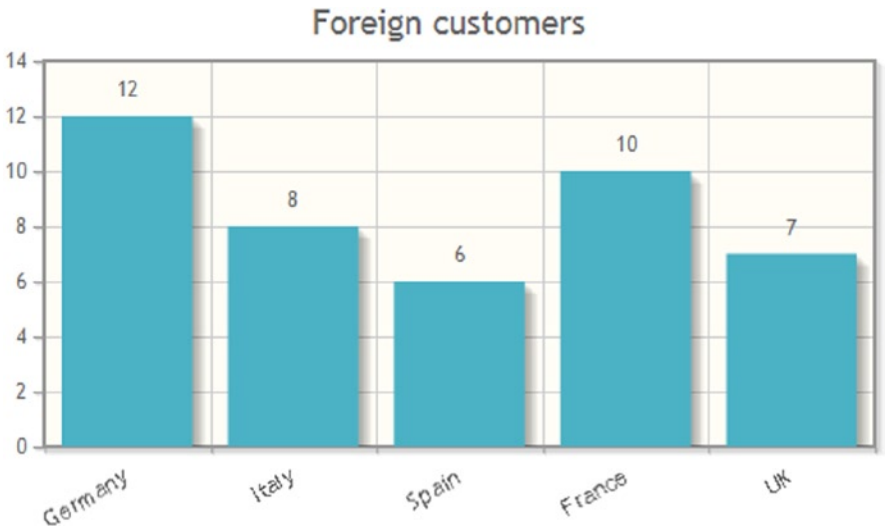


Figure 10-4. *A bar chart reporting the y value above each bar*

Bars with Negative Values

Generally, we are accustomed to seeing bar charts with all positive y values, but this is not always the case. If you want to represent negative values on a bar chart, however, you must be careful. If you try to use an input data array containing negative values, as in the following example

```
var data = [['Germany', -12], ['Italy', -8], ['Spain', -6], ['France', -10], ['UK', -7]];
```

you get the bar chart in Figure 10-5.

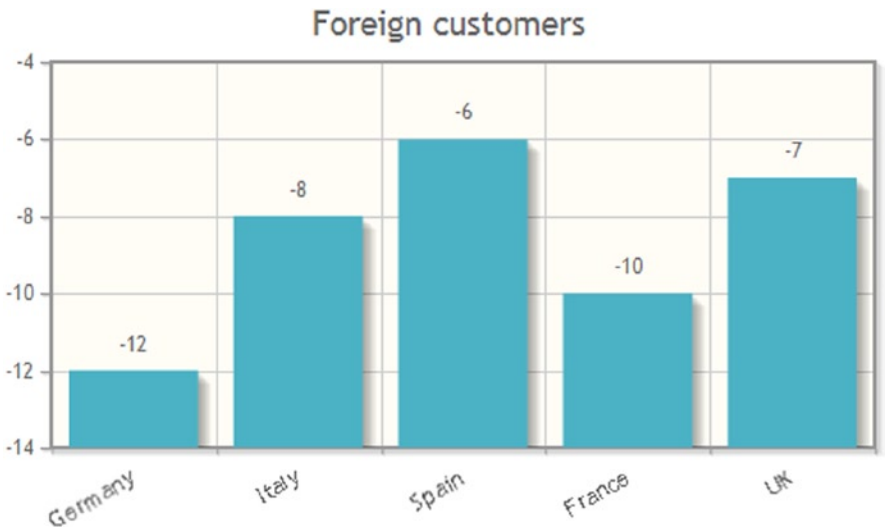


Figure 10-5. *This bar chart has interpreted the negative values badly*

This is not what you really wanted. The bars are drawn as if they were still positive. Only the point labels show the values of y properly. Furthermore, the values reported on the y axis are not correlated with the representation of the bars, which should start at the top and go down to the corresponding negative value on the y axis. To overcome all these issues, you need to set the `fillToZero` property to `'true'`, a property belonging to the *BarRenderer* plug-in; therefore, you have to specify this in `rendererOptions` (see Listing 10-5).

Listing 10-5. ch10_04a.html

```
var options = {
    title: 'Foreign Customers',
    seriesDefaults:{
       renderer:$.jqplot.BarRenderer,
       rendererOptions: { fillToZero: true },
       pointLabels: { show: true }
    },
    ...
```

Now, jqPlot can represent the negative bars correctly (see Figure 10-6).

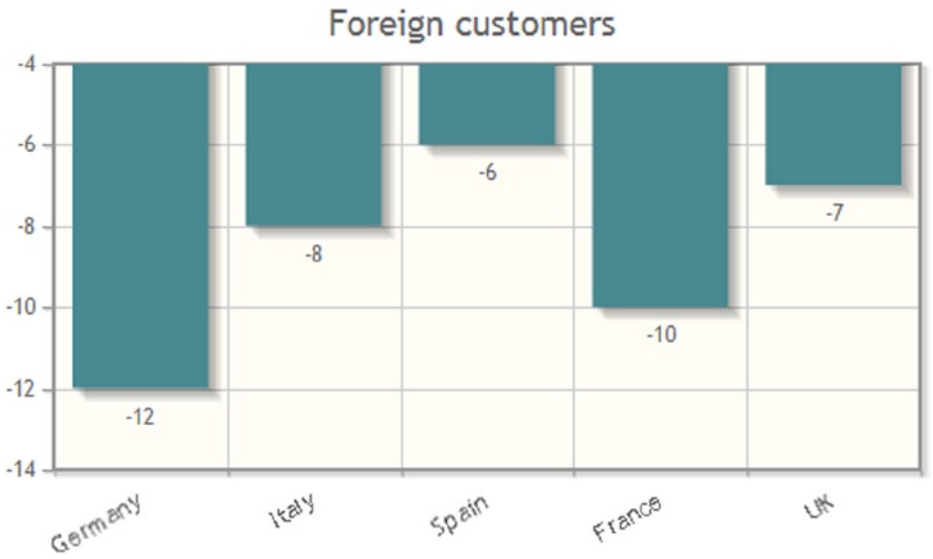


Figure 10-6. *A simple bar chart with negative values*

This functionality is better appreciated when both positive and negative values are represented in the same bar chart:

```
var data = [['Germany', -12], ['Italy', 8], ['Spain', -6], ['France', 10], ['UK', -7]];
```

As you can see in Figure 10-7, a slightly darker color distinguishes the bars with negative values.

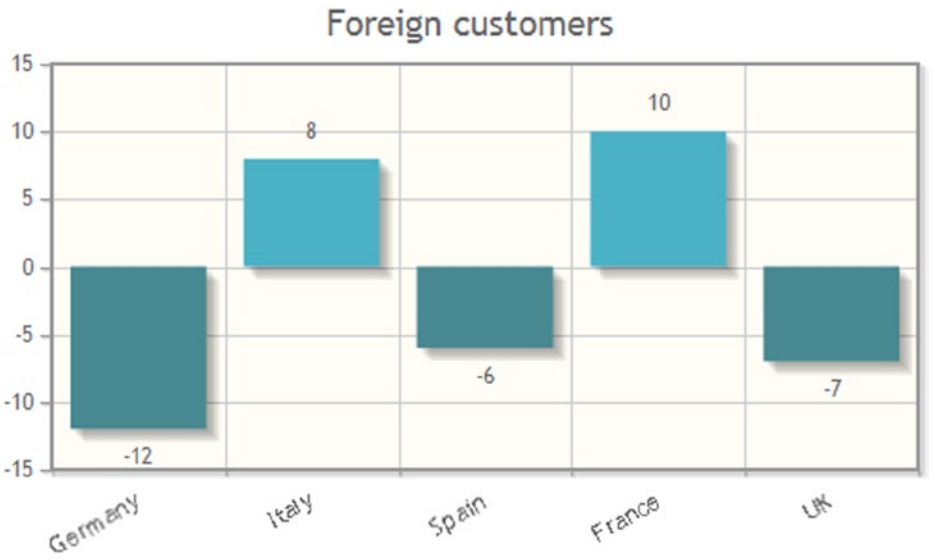


Figure 10-7. *A simple bar chart with positive and negative values*

Bar Charts with More Than One Set of Data

You have seen how line charts manage multiple series, and so you might expect bar charts to afford the same possibility. In the transition from single series to multiple series, you need to make some changes in the way input data are organized. Hence, you start from the format for the input data array for a single series:

```
var data = [['Germany', 12], ['Italy', 8], ['Spain', 6], ['France', 10], ['UK', 7]];
```

First, you have to specify a customized `ticks` array, which must contain the names of the groups, or categories, of data (the values you want to report on the x axis). The number of ticks should match the number of y values in each series.

```
var ticks = ['Germany', 'Italy', 'Spain', 'France', 'UK'];
```

Because you are working with more than one series, you can specify at least three series of data. Each series represents a further classification of the data, so you can distinguish one from the other by means of a label reporting the group the data belong to. Now, you have to insert only y values for each series (from the `ticks` array), as in Listing 10-6, given that the x values are the same for each.

Listing 10-6. ch10_05.html

```
var data = [12, 8, 6, 10, 7];     // Electronics customers
var data2 = [14, 12, 4, 14, 11];  // Software customers
var data3 = [18, 10, 5, 9, 9];    // Mechanics customers
```

With regard to the names indicating the series, you must specify these within the `series` object (see Listing 10-7), assigning them one by one to the `label` property of each series.

Listing 10-7. ch10_05.html

```
var options = {
        title: 'Foreign Customers',
        seriesDefaults:{
            renderer:$.jqplot.BarRenderer,
        },
    series:[
       {label: 'Electronics'},
       {label: 'Software'},
       {label: 'Mechanics'}
    ],
    axes: {
    ...
```

Now, as always within `options`, you assign the `ticks` array to the `ticks` property of the `xaxis` object (see Listing 10-8). In so doing ,you have assigned each tick generated on the x axis to a string contained in the array.

Listing 10-8. ch10_05.html

```
axes: {
    xaxis: {
       renderer: $.jqplot.CategoryAxisRenderer,
       ticks: ticks
    }
},
```

You have just seen that working with multiple series simply adds a further categorization of data. In addition to being divided into categories represented on the x axis, the data are divided into multiple series, each representing a different group. To distinguish one series from another, it is necessary to draw the corresponding bars with different colors. But, if you were to stop there, the user observing this chart would not have any information regarding which group is indicated by which color. The introduction of a legend is therefore required (see Listing 10-9).

Listing 10-9. ch10_05.html

```
var options = {
    title: 'Foreign Customers',
    seriesDefaults: {
       renderer: $.jqplot.BarRenderer,
    },
    series:[
       {label: 'Electronics'},
       {label: 'Software'},
       {label: 'Mechanics'}
    ],
    axes: {
       xaxis: {
          renderer: $.jqplot.CategoryAxisRenderer,
          ticks: ticks
       }
    },
```

```
    legend: {
       show: true,
       placement: 'outsideGrid',
       location: 'e'
    }
};
$.jqplot ('myChart', [data, data2, data3], options);
```

You thus obtain a multiseries bar chart, as illustrated in Figure 10-8. As you can see, when you use multiple series, you must use different colors in order to distinguish between them.

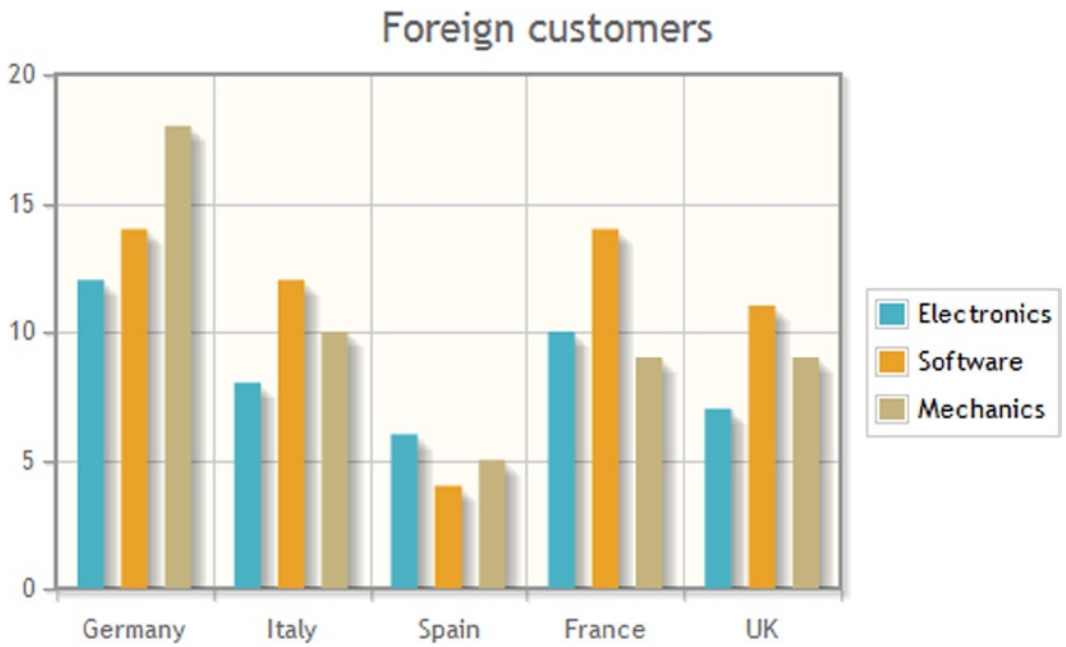


Figure 10-8. *A multiseries bar chart containing a legend*

Vertical and Horizontal Bar Charts

Looking at the chart in Figure 10-8, you will note that each group is represented by a color. The assigned colors follow the sequence specified internally to jqPlot by default, and that sequence is reflected in the line chart. Each country has three columns represented in its segment on the x- axis, with every segment bounded by grid lines.

This kind of bar chart is generally defined as a vertical bar chart. Nothing prevents us from representing the same input data with bars oriented horizontally, but here, too, you will need to make changes in the format of the input data arrays. In this case, it is necessary to use [y, n] pairs, where n is an integer value that is assigned to a string (see Listing 10-10). The strings are the label descriptions contained in the `ticks` array, and n is its index.

Listing 10-10. ch10_06.html

```
var data =  [[12, 1], [8, 2], [6, 3], [10, 4], [7, 5]];
var data2 =  [[14, 1], [12, 2], [4, 3], [14, 4], [11, 5]];
var data3 =  [[18, 1], [10, 2], [5, 3], [9, 4], [9, 5]];
var ticks = ['Germany', 'Italy', 'Spain', 'France', 'UK'];
```

After you have changed the format of the input data arrays, you must set the `barDirection` property to `'horizontal'` (`'vertical'` is the default value). As shown in Listing 10-11, you have to do so in `seriesDefaults` in order to apply the horizontal orientation to all series. This time, it is necessary to assign the `ticks` array to the `ticks` property in the `yaxis` object, instead of the `xaxis` object, as before.

Listing 10-11. ch10_06.html

```
var options = {
    title: 'Foreign Customers',
    seriesDefaults:{
       renderer: $.jqplot.BarRenderer,
       rendererOptions: {
          barDirection: 'horizontal'
       }
    },
    series:[
       {label: 'Electronics'},
       {label: 'Software'},
       {label: 'Mechanics'}
    ],
    axes: {
       yaxis: {
          renderer: $.jqplot.CategoryAxisRenderer,
          ticks: ticks
       }
    },
    legend: {
       show: true,
       placement: 'outsideGrid',
       location: 'e'
    }
};

$.jqplot ('myChart', [data, data2, data3], options);
```

Now, you get the horizontal multiseries bar chart shown in Figure 10-9.

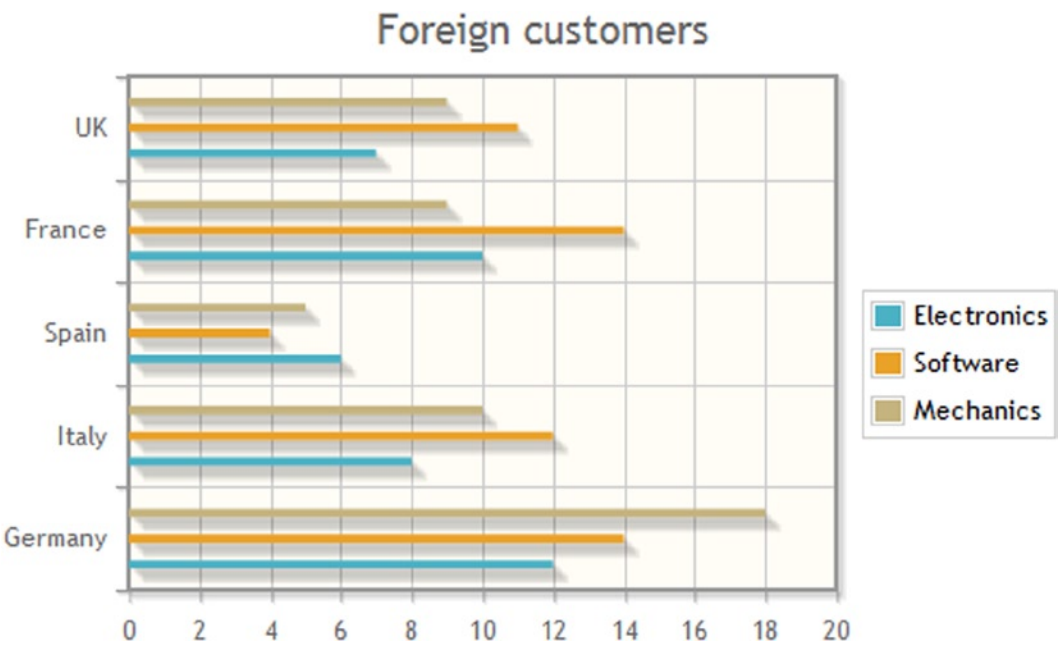


Figure 10-9. *A horizontal multiseries bar chart*

Vertical Stacked Bars

When you need to break down data series into their constituent parts, while retaining the ability to compare these data series as a whole, you have to use a stacked chart. jqPlot library supports such charts. With stacked bar charts, it is especially appropriate to add point labels, to make the chart more readable. The values reported are cumulative, that is, the sum of the underlying bars in the stack, as in Listing 10-12.

Listing 10-12. ch10_07.html

```
var data = [12, 8, 6, 10, 7];
var data2 = [14, 12, 4, 14, 11];
var data3 = [18, 10, 5, 9, 9];
var ticks = ['Germany', 'Italy', 'Spain', 'France', 'UK'];

 var options = {
   title: 'Foreign Customers',
   stackSeries: true,
   seriesDefaults:{
      renderer:$.jqplot.BarRenderer,
      pointLabels: { show: true,location: 's' }
   },
```

```
    series:[
        {label: 'Electronics'},
        {label: 'Software'},
        {label: 'Mechanics'}
    ],
    axes: {
        xaxis: {
            renderer: $.jqplot.CategoryAxisRenderer,
            ticks: ticks
        }
    },
    legend: {
        show: true,
        placement: 'outsideGrid',
        location: 'e'
    }
};

$.jqplot ('myChart', [data, data2, data3], options);
```

By setting values for the `pointLabels` property, you can specify the location where the point labels are shown: `'n'`, `'s'`,` e'`, `'w'`, `'ne'`, `'nw'`, `'se'`, or `'sw'`. These values should be interpreted as the cardinal points indicating the direction in which to draw the point label, with respect to the top of the bar. In this example, you choose `'s'` (south) to display the value just below the top of the bar, in the colored area, as presented in Figure 10-10.

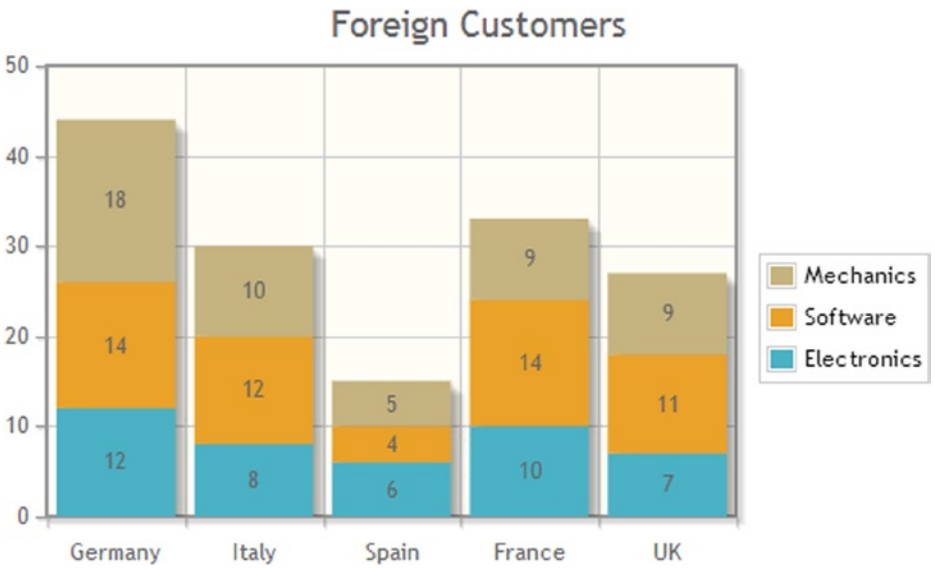


Figure 10-10. *A vertical multiseries stacked bar chart*

Horizontal Stacked Bars

In the same way, you can create a horizontal stacked bar chart. In this case, in order to represent the point labels inside the segments, you need to set them to `'w'` (west) (see Listing 10-13).

Listing 10-13. ch10_08.html

```
var data =  [[12, 1], [8, 2], [6, 3], [10, 4], [7, 5]];
var data2 =  [[14, 1], [12, 2], [4, 3], [14, 4], [11, 5]];
var data3 =  [[18, 1], [10, 2], [5, 3], [9, 4], [9, 5]];
var ticks = ['Germany', 'Italy', 'Spain', 'France', 'UK'];

var options = {
   title: 'Foreign Customers',
   stackSeries: true,
   seriesDefaults:{
      renderer: $.jqplot.BarRenderer,
      rendererOptions: {
         barDirection: 'horizontal'
      },
      pointLabels: { show: true, location: 'w' }
   },
   series:[
      {label: 'Electronics'},
      {label: 'Software'},
      {label: 'Mechanics'}
   ],
   axes: {
      yaxis: {
         renderer: $.jqplot.CategoryAxisRenderer,
         ticks: ticks
       }
   },
   legend: {
   show: true,
   placement: 'outsideGrid',
   location: 'e'
   }
};

$.jqplot ('myChart', [data, data2, data3], options);
```

In Figure 10-11, you can see how the values are shown near the end of the bar (west), in the colored area.

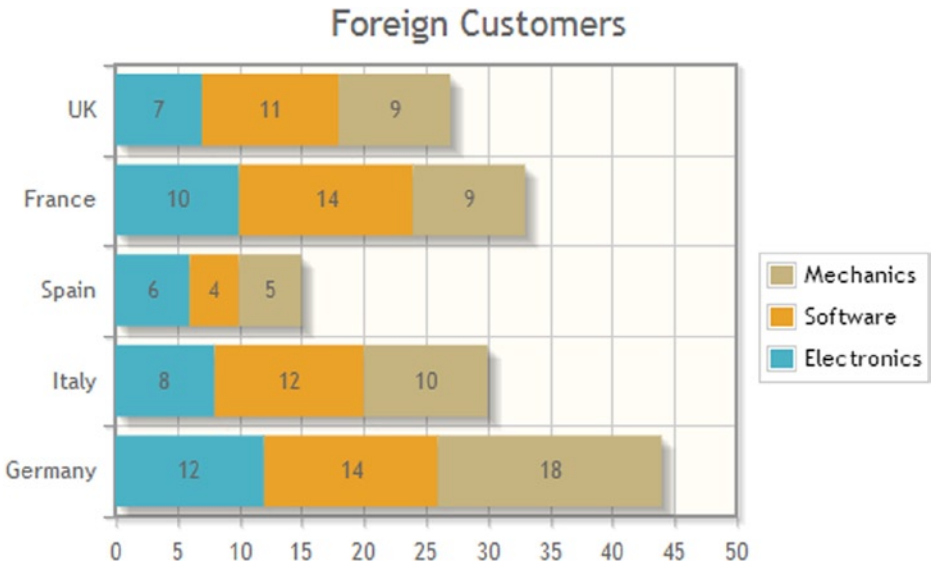


Figure 10-11. *A horizontal multiseries stacked bar chart*

Combination Charts: Lines in Bar Charts

A combination chart is a chart that combines two or more chart types in a single chart. In the following example, you will consider a bar chart series and a line chart series represented at the same time in one chart. For this kind of representation, you need to use a dual y axis. Each series has its own unit and magnitude and so must conform to one of these axes. Therefore, you have to use primary and secondary axes. You have to activate the autoscale functionality, too, in order to force the y axes to line up tick marks, thus obtaining consistent grid lines.

Let us, therefore, define the two input series (see Listing 10-14). The array `data` contains the [label1, y1] pairs of values to be presented as a bar chart. The array `line` contains the [label2, y2] pairs of values to be presented as a line chart.

Listing 10-14. ch10_09.html

```
var data = [['Germany', 12], ['Italy', 8], ['Spain' ,6], ['France', 10], ['UK', 7]];
var line = [['BMW', 45], ['AlfaRomeo', 30], ['Seat', 24],['Renault', 36], ['Mini', 30]];
```

This case is useful for understanding the utility of having multiple y axes to work on (jqPlot supports up to nine y axes and two x axes). Here, you have two series, the sequences of which are defined by the order in which you pass them as a second argument in the function `$.jqplot()`:

```
$.jqplot ('myChart', [data, line], options);
```

The array `data`, the series intended for the bar chart, is first, and the array `line`, intended for the line chart, is second. This is very important. Having established this order, in `options`, you need to specify two elements within the `series` object, as shown in Listing 10-15. In the first element only, you activate the *BarRenderer* plug-in, whereas in the second, you define two supplementary axes: `x2axis` and `y2axis`. Now, you have four axes to work with, and, consequently, you have to specify them within the `axes` object. On `yaxis` and `y2axis`, you must activate autoscale.

Listing 10-15. ch10_09.html

```
var options = {
    title: 'Foreign customers',
    series:[{renderer: $.jqplot.BarRenderer},
       {
          xaxis: 'x2axis',
          yaxis: 'y2axis'
       }],
    axes: {
       xaxis: {
          renderer: $.jqplot.CategoryAxisRenderer
       },
       x2axis: {
          renderer: $.jqplot.CategoryAxisRenderer
       },
       yaxis: {
          autoscale: true
       },
       y2axis: {
          autoscale: true,
          renderOptions: {
             alignTicks: true
          }
       }
    }
};
```

The result is the chart in Figure 10-12, containing bars and lines at the same time.

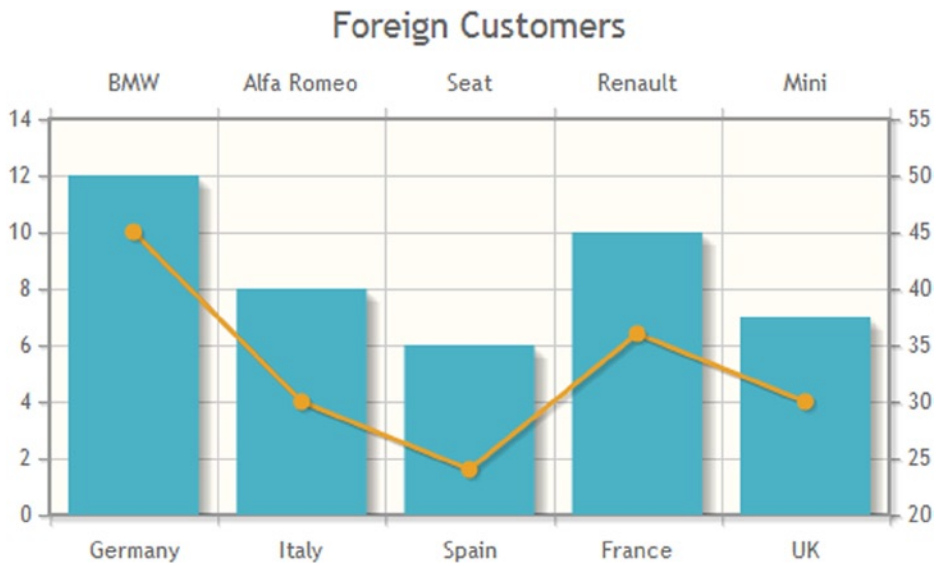


Figure 10-12. *A line chart combined with a bar chart*

Animated Plot

The jqPlot library also provides you with the ability to animate your charts. To this end, you do not need any further plug-ins. Starting from the previous example, the combination chart (see Listing 10-15), you can assign a different speed for each series. In defining a drawing speed, it is as though you were slowing down the creation of an element of the chart by the browser. This produces a dynamic effect during the drawing process, thus creating an animation. Furthermore, by assigning different speeds to different parts, you can obtain very nice effects.

As we can see in Listing 10-16, in `options`, you have to activate the animation functionality by setting to `'true'` the `animate` and the `animateReplot` properties. Then, you define a different speed for each series, using a numeric value (number of milliseconds).

Listing 10-16. ch10_10.html

```
var options = {
    animate: true,
    animateReplot: true,
    title: 'Foreign Customers',
    series:[{
       renderer: $.jqplot.BarRenderer,
       rendererOptions: {
          animation: {
             speed: 2500
          },
       }
    },{
       xaxis: 'x2axis',
       yaxis: 'y2axis',
       rendererOptions: {
          animation: {
             speed: 2500
          },
       }
    }],
    axes: {
       xaxis: { renderer: $.jqplot.CategoryAxisRenderer  },
       x2axis: {renderer: $.jqplot.CategoryAxisRenderer  },
       yaxis: { autoscale:true, numberTicks: 6 },
       y2axis: { autoscale:true, numberTicks: 6 }
    }
};
```

When you load this chart in a browser, you obtain an animation in which a line chart and a bar chart are drawn slowly and smoothly. Figure 10-13 shows how the animation develops in successive stages. The line chart is drawn from left to right, following the order of the data points, and, simultaneously, the bars grow to reach their respective y values.

Figure 10-13. *An animated combined line–bar chart*

Marimekko Chart

A kind of chart that can be derived from the bar chart is the so-called Marimekko chart (also called Mekko chart), named for its resemblance to a Marimekko print. This type of chart has been adopted in the business world. Marimekko charts are essentially stacked column charts. Here, however, all the bars are of equal height. Moreover, there are no spaces between the bars, and the bars are divided into several segments, the height of which is correlated to a percentage (see Figure 10-14).

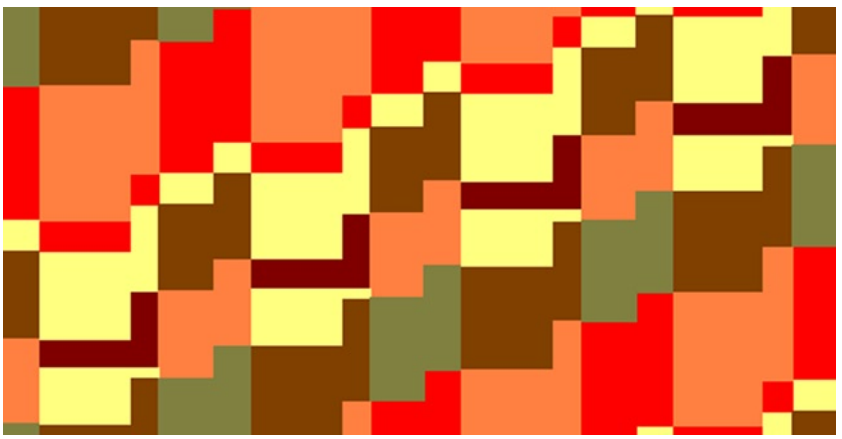


Figure 10-14. *A Marimekko pattern*

Marimekko charts are designed to report percentage values on both axes: the percentage covered by each category along the x axis, where each bar is placed, and the percentage covered by each category along the y axis, represented by the segments into which each bar is divided.

jqPlot allows you to develop this kind of chart, using two specific plug-ins: *MekkoRenderer* and *MekkoAxisRenderer*:

```
<script class="include" type="text/javascript"
     src="../src/plugins/jqplot.mekkoRenderer.min.js"></script>
<script class="include" type="text/javascript"
    src="../src/plugins/jqplot.mekkoAxisRenderer.min.js"></script>
<script class="include" type="text/javascript"
    src="../src/plugins/jqplot.canvasTextRenderer.min.js"></script>
<script class="include" type="text/javascript"
    src="../src/plugins/jqplot.canvasAxisLabelRenderer.min.js"></script>
```

Or, if you prefer to use a CDN service, you may do so as follows:

```
<script type="text/javascript" src="http://cdn.jsdelivr.net/jqplot/1.0.8/plugins↩
    /jqplot.mekkoRenderer.min.js"></script>
<script type="text/javascript" src="http://cdn.jsdelivr.net/jqplot/1.0.8/plugins↩
    /jqplot.mekkoAxisRenderer.min.js"></script>
<script type="text/javascript" src="http://cdn.jsdelivr.net/jqplot/1.0.8/plugins↩
    /jqplot.canvasTextRenderer.min.js"></script>
<script type="text/javascript" src="http://cdn.jsdelivr.net/jqplot/1.0.8/plugins↩
    /jqplot.canvasAxisLabelRenderer.min.js"></script>
```

Along with these two plug-ins, you need to include *CanvasTextRenderer* and *CanvasAxisLabelRenderer*. Data are specified for each bar in the chart. You can specify data as an array of y values or as an array of [label, value] pairs. In Listing 10-17, note that labels are used only for the first series; labels for subsequent series will be ignored.

Listing 10-17. ch10_11.html

```
var bar1 = [['bananas', 10],['apples', 7],['pears', 4],
            ['peaches', 8],['lemons', 7],['oranges',5]];
var bar2 = [9, 5, 8, 11, 9, 4];
var bar3 = [11, 4, 7, 3, 8, 7];
var bar4 = [5, 8, 11, 4, 12, 3];
var barLabels = ['Italy', 'Spain', 'France', 'Greece'];
```

In `options`, you activate the *MekkoRenderer* plug-in, assigning it to the `seriesDefaults` object. You can add a legend on the right side of the chart by setting the `show` property of the `legend` object to `'true'`. If you want to place labels for each bar below the x axis, you must assign the `barLabels` array to the `barLabels` property on the x axis.

Listing 10-18. ch10_11.html [no callout]

```
var options = {
    title: 'Fruit Consumption in 2012',
    seriesDefaults:{renderer: $.jqplot.MekkoRenderer},
    legend:{show: true},
    axesDefaults:{
       renderer: $.jqplot.MekkoAxisRenderer
    },
```

```
    axes:{
       xaxis:{
          barLabels: barLabels,
          tickOptions:{formatString: '%d'}
       }
    }
};

$.jqplot('myChart', [bar1, bar2, bar3, bar4], options);
```

Now, you get the Mekko chart illustrated in Figure 10-15.

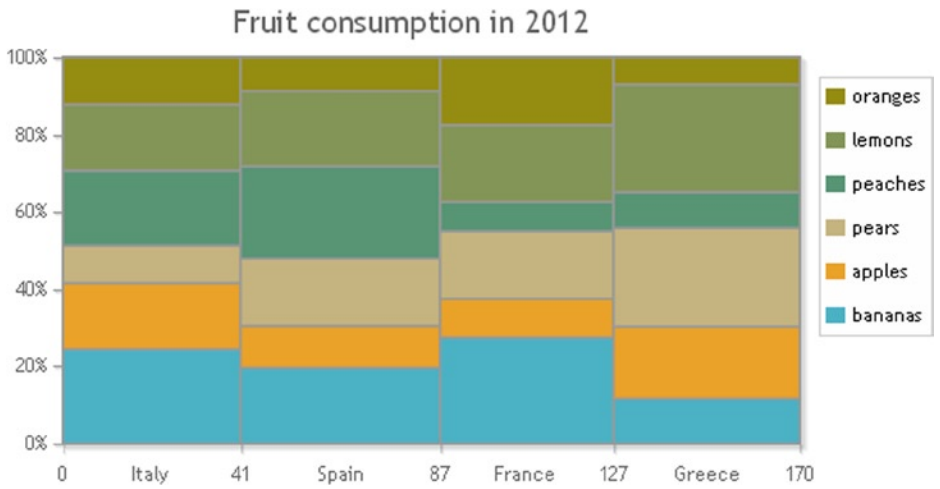


Figure 10-15. *A Mekko chart*

Bar Chart Events

In a bar chart, if you move the cursor over a bar, it will be highlighted by default. Events are triggered when you mouse over a bar and also when you click a bar. The ability to capture these events and to manage them is very important, and the jqPlot library allows you to do so. You can implement a specific response action for different types of events, thus making your chart much more interactive. The response you obtain can depend on where you find the mouse pointer or which targets you click.

Table 10-1 reports events that, owing to their wealth of elements, lend themselves to application in a bar chart. Let us take a look at such events one by one.

Table 10-1. *Handling Events with the jqPlot Library*

Event	When Triggered
`jqplotDataClick`	You click with the left mouse button on the data point.
`jqplotRightClick`	You click with the right mouse button on the data point.
`jqplotDataMouseOver`	You mouse over the data point.
`jqplotDataHighlight`	The data point is highlighted.
`jqplotDataUnhighlight`	The data point is unhighlighted.

The jqplotDataClick Event

This example presents the `jqplotDataClick` event—the clicked series index, the point, and its data values.

Let us start by considering the first example, the simple bar chart (see Listing 10-19).

Listing 10-19. ch10_12.html

```
var data = [['Germany', 12], ['Italy', 8], ['Spain', 6],
            ['France', 10], ['UK', 7]];

var options = {
   title: 'Foreign Customers',
   series:[{renderer: $.jqplot.BarRenderer}],
   axes: {
      xaxis: {
         renderer: $.jqplot.CategoryAxisRenderer
      }
   }
};

$.jqplot ('myChart', [data], options);
```

Inside the jQuery `ready()` function, you add the function in Listing 10-20. This is a jQuery function in which you bind the `jqplotDataClick` event with the execution of a function. As argument, this event takes some values of attributes of the jqPlot object, such as `seriesIndex`, `pointIndex`, and `data`. These values will be converted into a string and concatenated with the jQuery `html()` function. This HTML text will be sent to the `info1` element in the web page.

Listing 10-20. ch10_12.html

```
$('#myChart').bind('jqplotDataClick',
    function (ev, seriesIndex, pointIndex, data) {
       $('#info1').html('series: ' + seriesIndex +
       ', point: '+pointIndex+', data: '+data);
    }
);
```

Now, you add a `<span>` element where you want to show the text with values. This element will show a "Nothing yet" message until you click a bar. Then, a new text will replace the message with values, depending on the point clicked:

```
<div><span>You clicked: </span><span id="info1">Nothing yet</span></div>
```

Figure 10-16 shows the message corresponding to the event triggered when the user clicks the "France" bar.

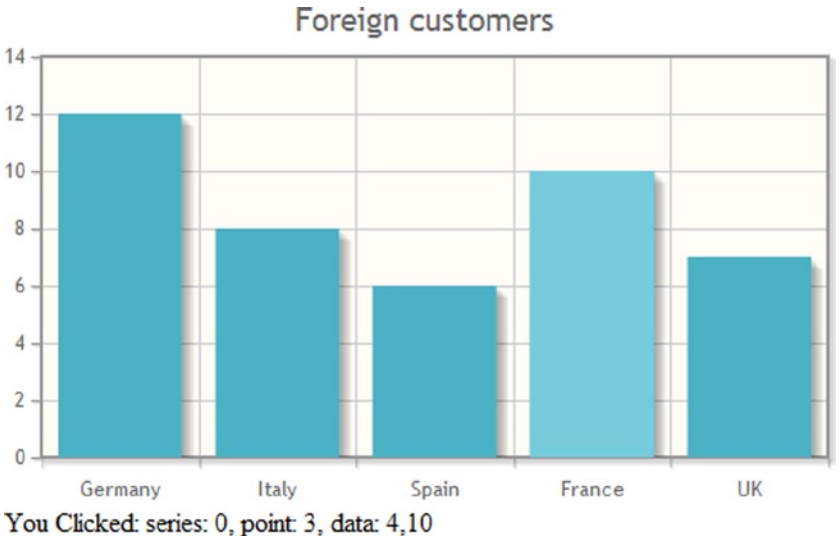


Figure 10-16. *By clicking a bar, you can obtain its values*

The jqplotRightClick Event

This example covers another event that jqPlot provides: `jqplotRightClick`. This event requires an explicit activation in `options` (see Listing 10-21). This causes jqPlot to fire a `jqplotRightClick` event when the user right-clicks a bar.

Listing 10-21. ch10_13.html

```
var options = {
    title: 'Foreign Customers',
    captureRightClick: true,
    series:[{renderer: $.jqplot.BarRenderer}],
    axes: {
       xaxis: {
          renderer: $.jqplot.CategoryAxisRenderer
       }
    }
};
```

Next, you need to replace the previous jqPlot function with the one in Listing 10-22.

Listing 10-22. ch10_13.html

```
$('#myChart').bind('jqplotDataRightClick',
    function (ev, seriesIndex, pointIndex, data) {
       $('#info1').html('series: ' + seriesIndex +
       ', point: '+pointIndex+', data: '+data);
    }
);
```

The general effect is the same, but this time you have to right-click instead of left-click. Right-clicking the "Spain" bar gives you the result in Figure 10-17.

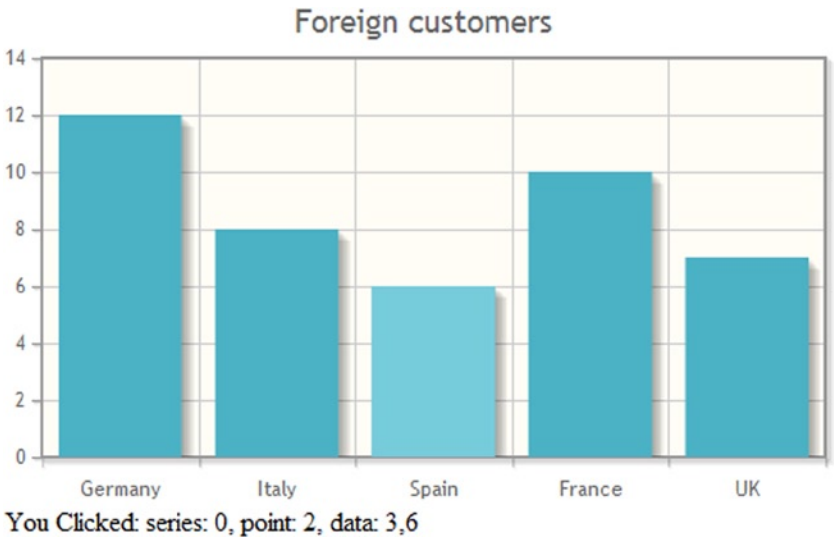


Figure 10-17. *By right-clicking a bar, you obtain its values*

Other Bar Chart Events

Often, you may want to capture another event: when the cursor is moused over a bar, jqPlot fires a `jqplotDataMouseOver` event. This event is also generated when you have explicitly disabled the highlighting. The event will fire continuously as the user passes the mouse over the bar. In contrast, another event, `jqplotDataHighlight`, fires only once, when the user first passes the mouse over the bar. When the user moves out of a bar, jqPlot fires a third event: `jqplotDataUnhighlight`. These last two events are generated only if highlighting is enabled.

Continuing from the previous example (see Listing 10-22), you replace the jQuery function that captures the `jqplotDataClick` event with two other functions (see Listing 10-23). The first will send an HTML text to the `info1` element with the same values as before as soon as you mouse over a bar. The second will replace the previous string in the `info1` element with `'Nothing'` just as you move out of the bar.

Listing 10-23. ch10_14a.html

```
$('#myChart').bind('jqplotDataHighlight',
    function (ev, seriesIndex, pointIndex, data) {
       $('#info1').html('series: ' + seriesIndex +
       ', point: '+pointIndex+', data: '+data);
    }
);

$('#myChart').bind('jqplotDataUnhighlight',
    function (ev) {
       $('#info1').html('Nothing');
    }
);
```

Now, you want to see the difference in behavior between the `jqplotDataMouseOver` event and the `jqPlotDataHighlight` event. There is no better way to understand this difference than with an example that allows you to compare them. This time, you will count the number of events fired when you mouse over a bar. To do this,

you define a counter, `nEvents`, initializing it to 0. As expected, with the `jqPlotDataHighlight` event, the counter is set to 1 whenever you mouse over a bar and assumes the value 0 when you move out of the bar. With the `jqplotDataMouseOver` event the behavior is very different: the counter increases continuously, keeping the cursor over the same bar. In both cases, you use the `jqplotDataUnhighlight` event to reset the counter every time you move out of a bar.

First, you need to change the `info1` HTML element:

```
<div><span>Events: </span><span id="info1">Nothing yet</span></div>
```

Then, to study the behavior of the `jqplotDataHighlight` event, you replace the two jQuery functions with the two in Listing 10-24.

Listing 10-24. ch10_14b.html

```
nEvents = 0;
$('#myChart').bind('jqplotDataHighlight',
    function (ev, seriesIndex, pointIndex, data) {
       nEvents = nEvents + 1;
       $('#info1').html(nEvents);
    }
);

$('#myChart').bind('jqplotDataUnhighlight',
    function (ev) {
       $('#info1').html('Nothing');
       nEvents = 0;
    }
);
```

As presented in Figure 10-18, while keeping the mouse on the "Spain" bar, the counter gives a steady 1.

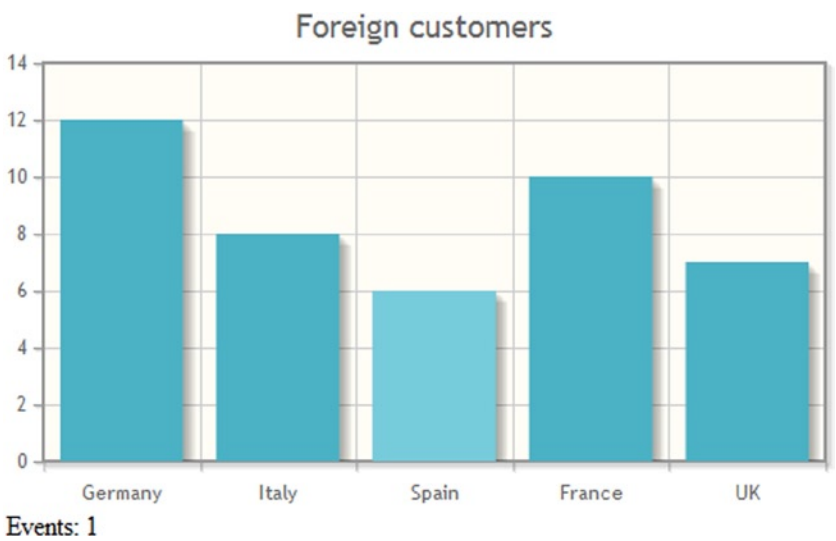


Figure 10-18. *Counting how many `jqPlotDataHighlight` events occur*

To study the behavior of the `jqplotDataMouseOver` event, you replace the two jQuery functions with the two in Listing 10-25.

Listing 10-25. ch10_14c.html

```
nEvents = 0;
$('#myChart').bind('jqplotDataMouseOver',
    function (ev, seriesIndex, pointIndex, data) {
       nEvents = nEvents + 1;
       $('#info1').html(nEvents);
    }
);

$('#myChart').bind('jqplotDataUnhighlight',
    function (ev) {
       $('#info1').html('Nothing');
       nEvents = 0;
    }
);
```

As shown in Figure 10-19, while keeping the mouse on the "Spain" bar, the counter continuously increases its value.

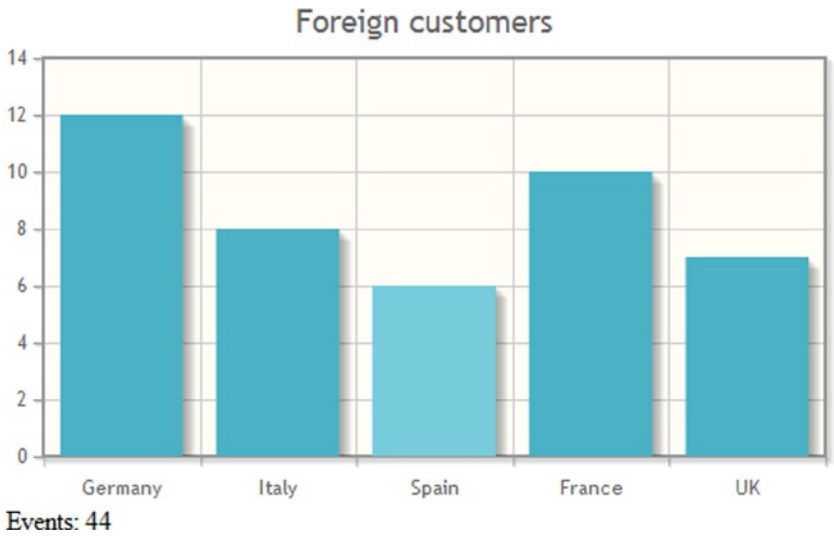


Figure 10-19. *Counting how many `jqplotDataMouseOver` events occur*

Clicking the Bar to Show Information in Text

Because of jqPlot's potential in managing events, let us take the opportunity to look at a common case. By clicking a bar, you can get information about that bar and show it in a text box on the HTML page. This is made possible by binding a listener to the `jqlotDataClick` event.

For this example, you start with the code used to generate the horizontal stacked bar chart (see Listing 10-26).

Listing 10-26. ch10_15.html

```
var data = [12, 8, 6, 10, 7];
var data2 = [14, 12, 4, 14, 11];
var data3 = [18, 10, 5, 9, 9];
var ticks = ['Germany', 'Italy', 'Spain', 'France', 'UK'];

var options = {
   title: 'Foreign Customers',
   stackSeries: true,
   seriesDefaults:{
      renderer: $.jqplot.BarRenderer,
      pointLabels: { show: true, location: 's' }
   },
   series:[
      {label: 'Electronics'},
      {label: 'Software'},
      {label: 'Mechanics'}
   ],
   axes: {
      xaxis: {
         renderer: $.jqplot.CategoryAxisRenderer,
         ticks: ticks
      }
   },
   legend: {
      show: true,
      placement: 'outsideGrid',
      location: 'e'
   }
};

$.jqplot ('myChart', [data, data2, data3], options);
```

As in the previous examples for events, in the end you add a jQuery function that captures the `jqplotDataClick` event and that sends a set of information to the `info1` element (see Listing 10-27).

Listing 10-27. ch10_15.html

```
$('#myChart').bind('jqplotDataClick',
    function (ev, seriesIndex, pointIndex, data) {
       $('#info1').html('series: ' + seriesIndex +
       ', point: '+pointIndex+', data: '+data);
    }
);
```

Whenever you click a bar, this function will refresh the information displayed where you have placed the `<span>` element with `'info1'` as `id`. Here, you add the `<span>` element to the HTML page:

```
<span id="info1">Information will be provided here </span>
```

Now, by clicking the highlighted area of a bar, you get all the data relative to that bar in the text box, as seen in Figure 10-20.

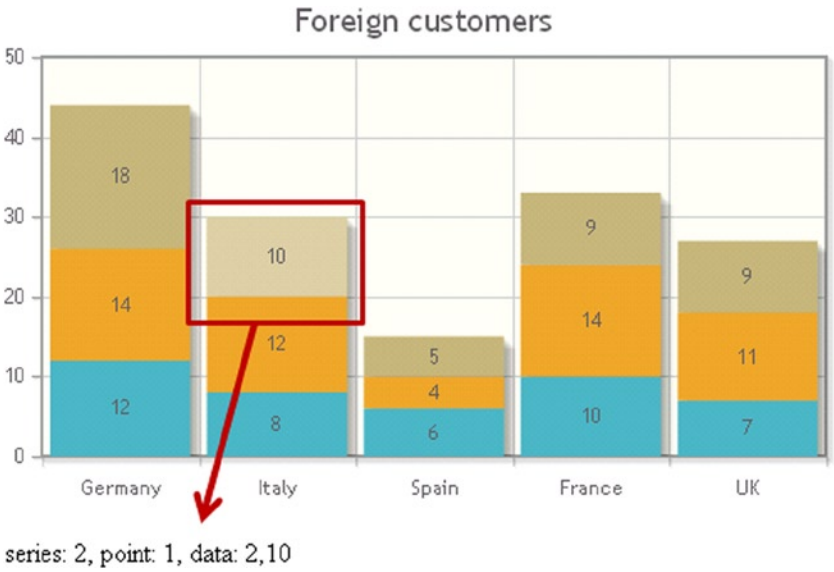


Figure 10-20. *By clicking a stacked bar, you obtain its values*

Handling Legends

Working with bar charts, you made use of the legend, a key component of most charts. The legend is a defined element within jqPlot. Usually, you need only call the `legend` object in `options` to make the legend pop up next to your chart. Here, you will analyze this useful element in more detail.

When does using a legend become necessary? When you are dealing with multiseries data, that is, when you have a grouping of data, often distinguished by different colors. The legend does nothing but report in a small space the relationship that exists between each color and a label distinguishing the elements of the group.

Adding a Legend

The previous example (see Figure 10-20) is perfect for studying legends. By observing the stacked chart, you can easily see that the bars for each country are made up of three portions, which are characterized by different colors. Thus, you understand that three series are represented in the chart. Furthermore, you also have information about the amount that each series contributes to the overall value, but still you lack crucial information: which categories are represented by which colors. By adding a legend, you will clarify the association between the three colors and these categories: "Mechanics," "Software," and "Electronics."

Therefore, continuing to work using the code from the previous example (see Listings 10-26 and 10-27), let us add the `legend` definition to the `options` object, as shown in Listing 10-28. To do this, you do not have to include any plug-ins; you need only set the `show` attribute to `'true'`.

Listing 10-28. ch10_15.html

```
var options = {
    ...
    axes: {
       xaxis: {
          renderer: $.jqplot.CategoryAxisRenderer,
          ticks: ticks
       }
    },
    legend: {
       show: true,
       placement: 'outsideGrid',
       location: 'e'
    }
};

$.jqplot ('myChart', [data,data2,data3],options);
```

Figure 10-21 illustrates the chart with a legend.

Here there will be the info

Figure 10-21. *A stacked bar chart with a legend*

The `placement` attribute specifies where you want the legend; omitting it, you get the default behavior: the legend is drawn inside the chart. To avoid covering the bars, you can change the position of the legend by setting the `location` attribute, as given in Listing 10-29.

Listing 10-29. ch10_16a.html

```
legend: {
    show: true,
}
```

Thus, as Figure 10-22 shows, the chart is changed, with the legend drawn inside (by default), in the top-right corner (`'ne'` [northeast]).

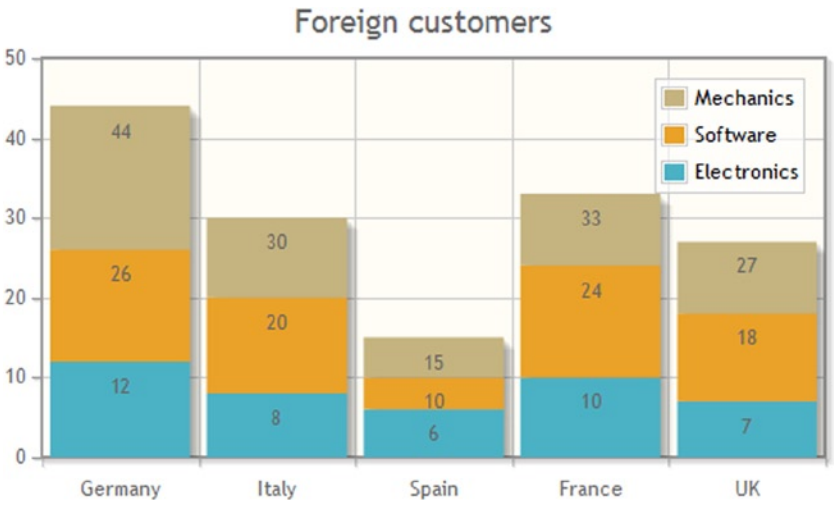


Figure 10-22. *The default legend position is inside the chart, in the top-right corner*

For a more thorough approach, it is best to use the Cascading Style Sheets (CSS) customization. You will use some CSS classes for the legend to modify the default attributes—mainly, the CSS class `table.jqplot-table-legend`.

For example, you can add the specifications to the CSS class offered in Listing 10-30.

Listing 10-30. ch10_16b.html

```
<style>
    table.jqplot-table-legend {
       background-color: rgba(175, 175, 175, 1);
       font: "Arial Narrow";
       font-style: italic;
       font-size: 13pt;
       color: white;
    }
</style>
```

And, the legend will change, as demonstrated in Figure 10-23.

Figure 10-23. *The modified legend, using CSS styles*

The Enhanced Legend

If you look at several plug-ins in the jqPlot distribution, you will find a plug-in related to legends: *EnhancedLegendRenderer*. This plug-in extends the functionalities of the legend: clicking the legend items, you can show or hide the corresponding series. You can see this with a concrete example. First, you include the plug-in in your web page:

```
<script type="text/javascript"
    src="../src/plugins/jqplot.enhancedLegendRenderer.min.js"></script>
```

Or, if you prefer to use a CDN service, you may do so as follows:

```
<script type="text/javascript" src="http://cdn.jsdelivr.net/jqplot/1.0.8/plugins➥
    /jqplot.enhancedLegendRenderer.min.js"></script>
```

Then, you must activate the plug-in in `options` in the same way you have done with other plug-ins, as presented in Listing 10-31.

Listing 10-31. ch10_16c.html

```
legend: {
    renderer: $.jqplot.EnhancedLegendRenderer,
    show: true,
    placement: 'outsideGrid',
    location: 'ne'
}
```

After you load the page in your browser, you get the chart in Figure 10-24. If you click an item from the legend, the corresponding series will disappear from the chart, leaving an empty space. This can be useful if you want to analyze only a subset of series, ignoring others. Let us therefore click "Software" in the legend and look at what happens.

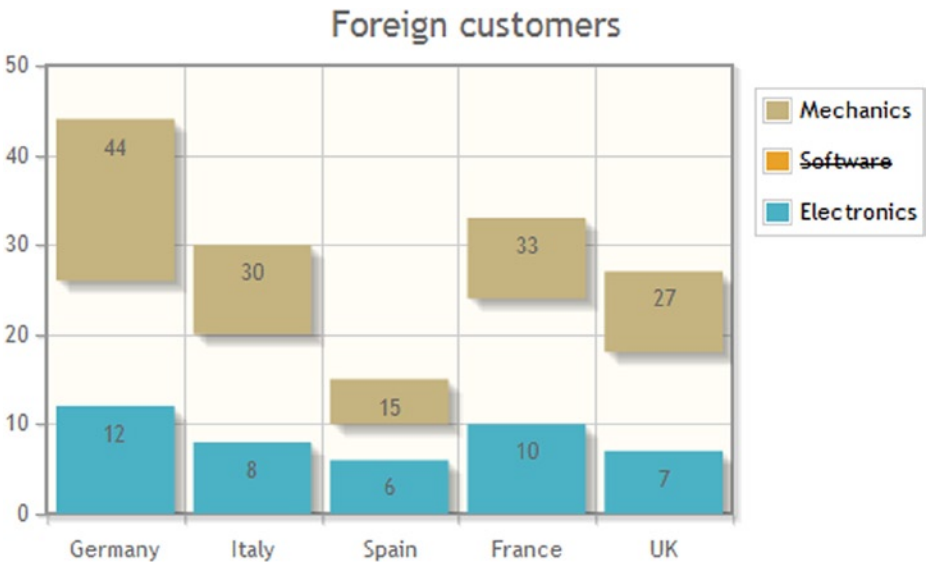


Figure 10-24. *You can hide a series by selecting an item in the legend*

In the legend in Figure 10-24, the orange segments belonging to Software series have disappeared, and the item "Software" is stricken out.

This effect is cumulative, and you can hide all the series, one by one. If you click a strikeout item again, the corresponding series will appear once more in the chart.

Custom Legend Highlighting

You have just seen how to use the legend that jqPlot provides by default. But, you can create a custom legend by implementing a simple table in HTML and then populating it dynamically, filling it with the labels for your series. Because you must start from scratch to create your own legend, it is first necessary to select a style. You can define the CSS style by using an external CSS file or writing the style directly in the web page, such as the one given in Listing 10-32.

Listing 10-32. ch10_17.html

```
<style type="text/css">
table.sample {
    border-width: thin;
    border-spacing: 0px;
    border-style: outset;
    border-color: rgb(221, 221, 221);
    border-collapse: collapse;
}
table.sample th {
    border-width: 1px;
    padding: 1px;
    border-style: inset;
    border-color: gray;
}
table.sample td {
    border-width: 1px;
    padding: 1px;
    border-style: inset;
    border-color: gray;
}
</style>
```

You have used three different CSS classes. The first specifies the style for the entire table. The other two are defined to specify a specific style for the headings and the cells, respectively.

After defining the style, for our purposes, you can use the code of a simple bar chart (see Listing 10-33).

Listing 10-33. ch10_17.html

```
var data = [['Germany', 12], ['Italy', 8], ['Spain', 6], ['France', 10], ['UK', 7]];

var options = {
    title: 'Foreign Customers',
    series:[{renderer:$.jqplot.BarRenderer}],
```

```
    axes: {
       xaxis: {
          renderer: $.jqplot.CategoryAxisRenderer
       }
    }
};

$.jqplot ('myChart', [data], options);
```

The next step to bind the custom legend to `jqplotDataHighlight` and `jqplotDataUnhighlight` events (see Listing 10-34). You have already seen these and how it is possible to bind an object to them using jQuery methods. In this case, you will do a lot more; you will ensure that the whole custom legend is dynamically created with only a few lines of jQuery. These will include the data array. Compared with the default legend offered by jqPlot, here you are able to obtain much more than the labels indicative of the groups' series membership. It is also possible to add summary values (using JavaScript functions) or simply the value of y.

Listing 10-34. ch10_17.html

```
$(document).ready(function(){

    var data = ...
    var options = ...

    $.jqplot ('myChart', [data], options);

    $.each(data, function(index, val) {
       $('#legend1').append('<tr><td>'+val[0]+'</td><td>'+val[1]+'</td></tr>');
    });

    $('#myChart').bind('jqplotDataHighlight',
       function (ev, seriesIndex, pointIndex, data) {
          var color = 'rgb(100%, 90%, 50%)';
          $('#legend1 tr').css('background-color', '#ffffff');
          $('#legend1 tr').eq(pointIndex+1).css('background-color', color);
    });

    $('#myChart').bind('jqplotDataUnhighlight',
       function (ev, seriesIndex, pointIndex, data) {
          $('#legend1 tr').css('background-color', '#ffffff');
    });

});
```

The first jQuery function handles the values of the data array. The other two bind the legend to the highlighting events when you mouse over the legend items. Moreover, these functions also bind a variation in style attributes to these events, in this case, the background color of the item.

Now, you need to define two different areas in which to insert the chart and the legend. You can accomplish this using an HTML table. So, you add the code in Listing 10-35 to the `<body>` section of the HTML page.

Listing 10-35. ch10_17.html

```
<table style="margin-left:auto; margin-right:auto;">
<tr>
    <td><div id="myChart" style="width:460px; height:340px;"></div></td>
    <td><div style="height:340px;">
       <table id="legend1" class="sample" >
          <tr><th>Nation</th><th>Customers</th></tr>
       </table>
    </div>
    </td>
</tr>
</table>
```

Finally, you can load the new page with the new custom legend (see Figure 10-25).

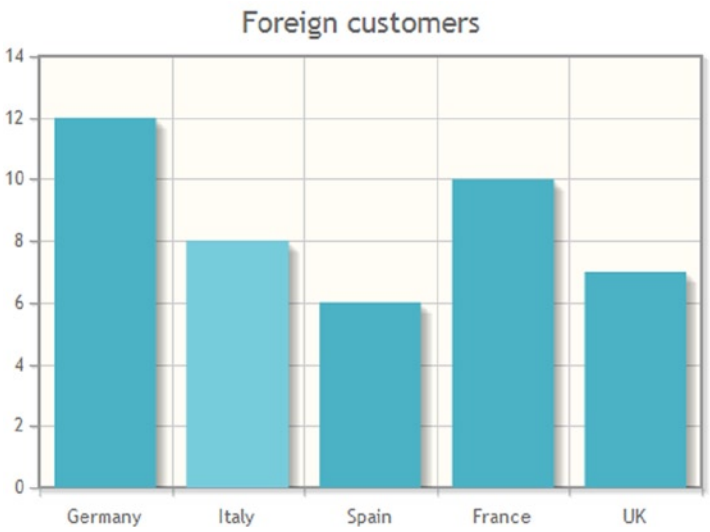


Figure 10-25. *A custom HTML legend*

Custom Tool Tip

In addition to legends, another very commonly used item in the bar chart is the tool tip. Just as there is the possibility to create a custom legend using code, it is also possible to customize tool tips, creating very original effects. When you mouse over a bar and highlight it, a tool tip will be shown, but, different from the default jqPlot tool tip, this will be totally built in HTML format. This greatly expands your possibilities of artistic expression, giving your chart a touch of personality (using jqPlot, everything risks appearing too "standard jqPlot"). In this example, you would like to show how a small icon image can give a very nice effect to a bar chart.

Before starting to write code, let us create a directory, and name it `flags` (you can name it as you prefer). In this directory you will store all the portable network graphics (PNG) image files you are going to use. These icons are flags of nations, which you will report on the x axis of the bar chart. It is very easy to find and download these PNG files from the Internet.

■ **Note** The PNG files required to show the flags in the tool tips are included in the source code accompanying the book, which you will find on the Source Code/Downloads tab of the book's Apress product page (www.apress.com/9781430262893).

When you have finished with the flag images, the first step is to create a custom tool tip. You need to bind tool tips to the `jqplotDataHighlight` and `jqplotDataUnhighlight` events. You can create custom tool tips dynamically with a few lines of jQuery.

Here, you start with the bar chart you have already used (see Listing 10-36), as it represents a simple example to understand the way to develop this kind of custom tool tip.

Listing 10-36. ch10_18.html

```
var data = [['Germany', 12], ['Italy', 8], ['Spain', 6], ['France', 10], ['UK', 7]];

var options = {
    title: 'Foreign Customers',
    series:[{renderer:$.jqplot.BarRenderer}],
    axes: {
       xaxis: {
          renderer: $.jqplot.CategoryAxisRenderer
       }
    }
};

$.jqplot ('myChart', [data], options);
```

You have to add two other arrays of data, containing some strings (see Listing 10-37). You will use these in the dynamically generated tool tip.

Listing 10-37. ch10_18.html

```
var tick = ['Germany', 'Italy', 'Spain', 'France', 'UK'];
var icon = ['germany.png', 'italy.png', 'spain.png', 'france.png', 'uk.png'];
```

You assign the return value of the `jqplot()` function to a variable, because you will need to use it later.

```
var myPlot = $.jqplot ('myChart', [data], options);
```

The jQuery code for binding the events to your custom tool tip is in Listing 10-38.

Listing 10-38. ch10_18.html

```
$('#myChart').bind('jqplotDataHighlight',
    function (ev, seriesIndex, pointIndex, data) {
       var chart_left = $('#myChart').offset().left;
       var chart_top = $('#myChart').offset().top;
       var x = data[0]*95+20;
       var y = myPlot.axes.yaxis.u2p(data[1]);
       var color = 'rgb(30%,50%,60%)';
```

```
        $('#tooltip1').css({left:chart_left+x, top:chart_top+y});
        $('#tooltip1').html('<span style="font-size:16px; font-weight:bold; color:' +
           color + ';">' + tick[data[0] - 1] +
           '</span><br/><img src="flags/'+ icon[data[0] - 1]+
           '" width="30" height="20"><br/> n:' + data[1]);
        $('#tooltip1').show();
    }
);

$('#myChart').bind('jqplotDataUnhighlight',
    function (ev, seriesIndex, pointIndex, data) {
        $('#tooltip1').empty();
        $('#tooltip1').hide();
    }
);
```

Finally, you must create two elements in the <body> section of the web page, as in Listing 10-39. In the first element, jqPlot will generate your custom tool tip on the canvas; in the second, jqPlot will create the canvas drawing the bar chart.

Listing 10-39. ch10_18.html

```
<div id="myChart" style="height:300px; width:500px;"></div>
<div id="tooltip1" style="position:absolute; height:0px; width:0px;"></div>
```

Figure 10-26 shows the "Germany" bar highlighted with the customized tool tip beside it. This will happen for all the bars in the bar chart, each showing the corresponding flag in the tool tip, every time the user highlights them by mousing over.

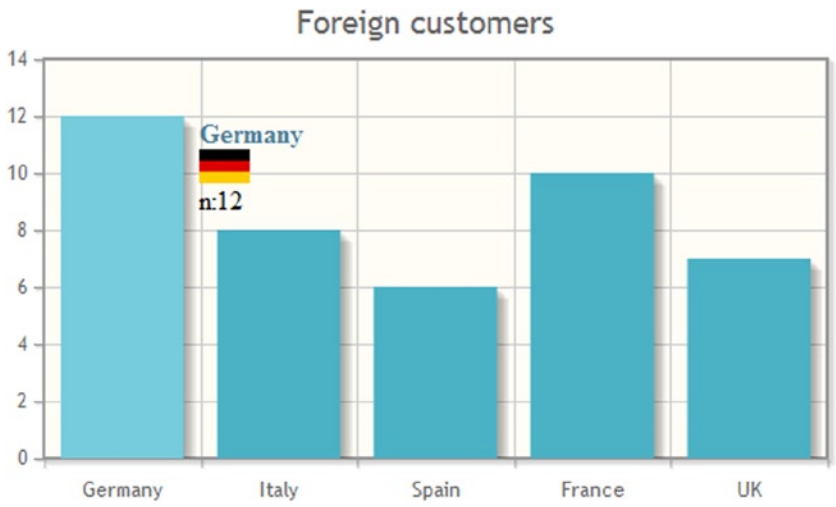


Figure 10-26. *A bar chart with custom tool tips*

Summary

In this chapter, you have seen how your data can be represented in a bar chart, using the *BarRenderer* plug-in. You began to see how, with the inclusion of this renderer plug-in, the structure of the main jqPlot object is gradually enriched with new properties and objects. Through practical examples, you learned how to change the values of property and object attributes with `rendererOptions`.

You also learned that it is sometimes possible to obtain different representations, using the same set of data. Knowing how to choose which representation is more suitable to your needs is one of the fundamental objectives of this book. For this purpose, you used the same set of data, both in a **grouped bar chart** and in a **stacked bar chart**. In both cases, you realized data representations with vertical and horizontal bars.

Later in this chapter, you looked at examples that demonstrated how jqPlot allows you to deal with **events**, using special functions. In addition, you further examined the **legend** component and the possibility of customizing legends using HTML code. Then, you applied the same approach to **tool tips**.

Often, the type of data that requires a bar chart representation may also be well represented by means of another type of chart: the **pie chart**. In the next chapter, you will discover how the jqPlot library handles this type of chart.

CHAPTER 11

Pie Charts and Donut Charts with jqPlot

Pie charts and donut charts are an excellent way to show the breakdown of data into their constituent parts. A pie chart is a circular chart divided into sectors, or "slices," and its main purpose is to illustrate their relative proportions: the arc length of each slice is proportional to the quantity it represents. A donut chart is very similar to a pie chart but has a hole in the center and supports the comparison of multiple series. In this chapter, you will look at both kinds of charts. The chapter concludes with a discussion of multidimentionsional pie charts.

Pie Charts

In jqPlot, data are interpreted as a line chart by default. If you want to show your data in a pie chart, you need to include the *PieRenderer* plug-in:

```
<script type="text/javascript" src="../src/plugins/jqplot.pieRenderer.min.js">
</script>
```

Or, if you prefer to use a content delivery network (CDN) service, you can do so as follows:

```
<script type="text/javascript" src="http://cdn.jsdelivr.net/jqplot/1.0.8/plugins
    /jqplot.pieRenderer.min.js"></script>
```

To better understand the use of this plug-in, let us take, for example, the amount of food a person consumes in a given period of time. This is a case in which a pie chart proves to be the best choice for data representation. All the food eaten makes up the whole group, and the various types of foods are the components you want to compare. Each type of food will be represented by a slice identified by means of a different color. The size of each slice will give a precise idea of the proportion that food type occupies in the diet of a person. You can start with a data array of [label, amount] pairs of values, as shown in Listing 11-1.

Listing 11-1. ch11_01a.html

```
var data = [ ['Dairy', 212],['Meat', 140], ['Grains', 276],
             ['Fish', 131],['Vegetables', 510], ['Fruit', 325] ];
```

Now, you can define the options. As you can see in Listing 11-2, you need to activate the plug-in and apply it to the `defaultSeries` object.

Listing 11-2. ch11_01a.html

```
var options = {
    seriesDefaults: {
       renderer: jQuery.jqplot.PieRenderer,
       rendererOptions: {
         showDataLabels: true
      }
   }
};

$.jqplot ('myChart', [data], options);
```

Figure 11-1 shows a simple pie chart without specification of additional attributes.

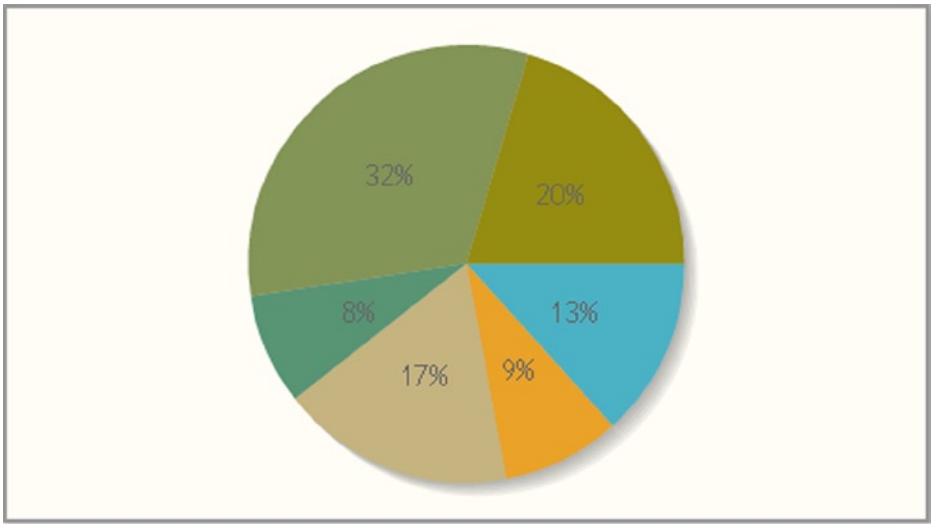


Figure 11-1. *A simple pie chart*

As demonstrated in Figure 11-1, inside its sectors the pie chart reports the percentage by default. If, instead, you want to display the value, as shown at the top right of Figure 11-2, you need to set `'value'` on the `dataLabels` property in `renderOptions`, as given in Listing 11-3.

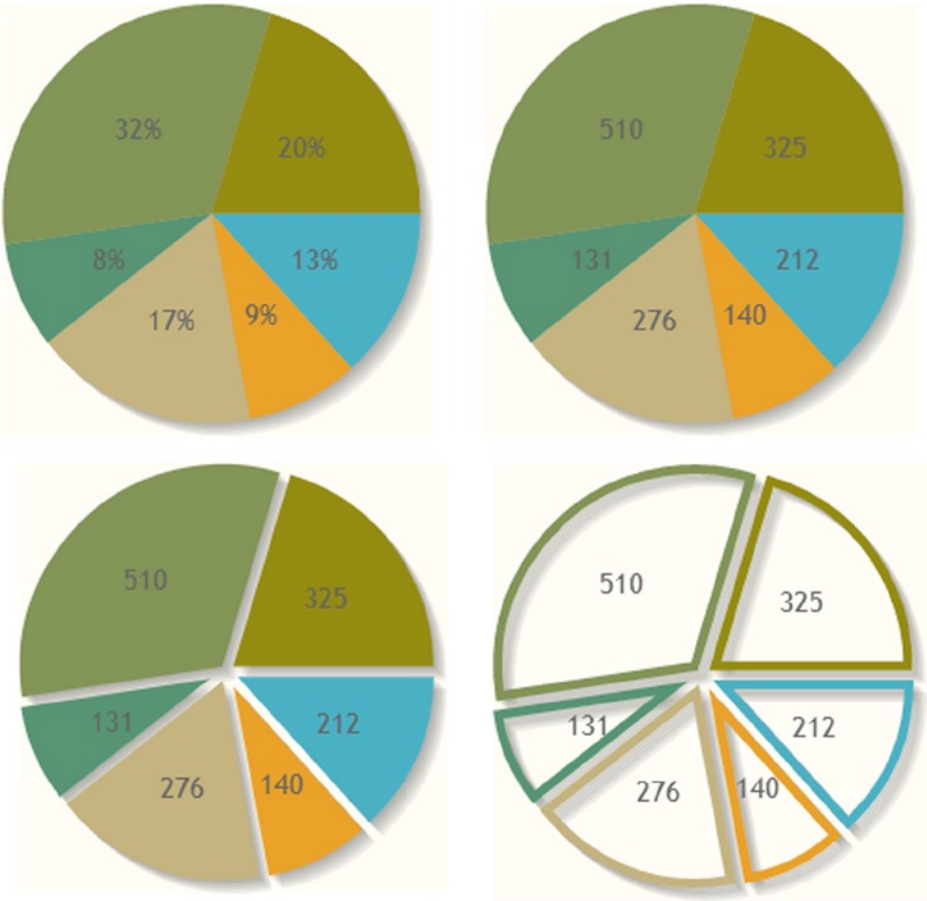


Figure 11-2. *Different ways to set a pie chart*

Listing 11-3. ch11_01b.html

```
rendererOptions: {
    showDataLabels: true,
    dataLabels: 'value'
}
```

If you want to add a margin to separate the slices of the pie, as illustrated in the lower-left of Figure 11-2, we need to set the `sliceMargin` property to 6 (see Listing 11-4).

Listing 11-4. ch11_01c.html

```
rendererOptions: {
    showDataLabels: true,
    dataLabels: 'value',
    sliceMargin: 6
}
```

Furthermore, if you want to show the pie chart with empty slices, as seen in the lower-right of Figure 11-2, you can set the `fill` property to `'false'` and the `lineWidth` property to 5 in order to shade the slices, with lines that are a little thicker (see Listing 11-5).

Listing 11-5. ch11_01d.html

```
rendererOptions: {
    showDataLabels: true,
    dataLabels: 'value',
    sliceMargin: 6,
    fill: false,
    // stroke the slices with a little thicker line.
    lineWidth: 5
}
```

Donut Charts

One of the main problems with pie charts is their inability to display multiple series simultaneously. Hence, you must decide whether to represent each series separately, with a pie chart or, preferably, to use a donut (also spelled "doughnut") chart. This kind of chart requires and uses options that are identical to those of pie charts; therefore, the transition from pie chart to donut chart is immediate.You will look at the simplicity of such a transition with an easy example.

First, as with the pie chart, you need to include a specific plug-in in order to obtain a donut chart:

```
<script type="text/javascript" src="../src/plugins/jqplot.donutRenderer.min.js"> </script>
```

The only change you have to make in the `options` object is to replace the `pieRenderer` object with `DonutRenderer` in the renderer call and then to modify the starting angle of the first sector in the `rendererOptions` object. By default the chart starts on the left side of the circle, but normally it must start at the top. So, it is necessary to set the `startAngle` property to–90 degrees (see Listing 11-6).

Listing 11-6. ch11_02.html

```
var options = {
    seriesDefaults: {
    // Make this a pie chart.
       renderer:$.jqplot.DonutRenderer,
       rendererOptions: {
          showDataLabels: true,
          dataLabels: 'value',
          sliceMargin: 3,
          startAngle: -90
       }
    }
};
jQuery.jqplot ('myChart', [data], options);
```

In this way, you get a donut chart (see Figure 11-3), which is very similar to the pie chart in the lower right of Figure 11-2.

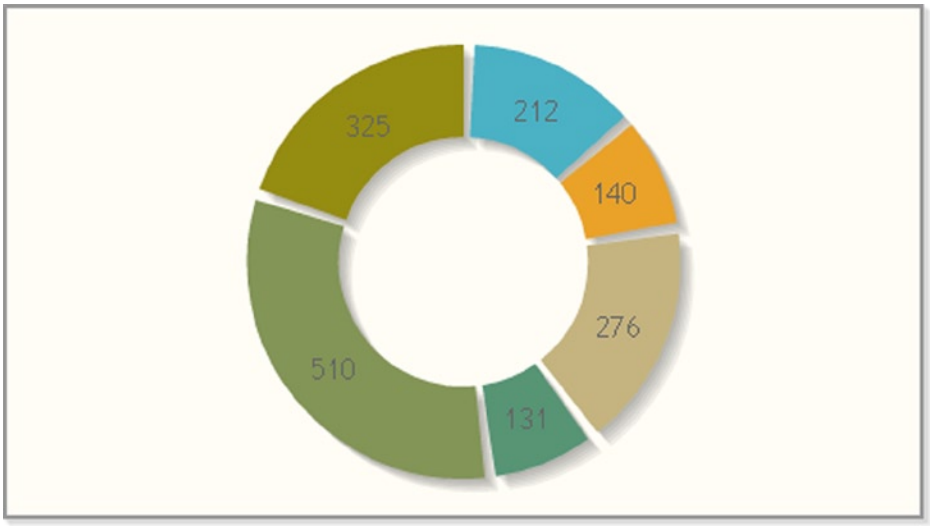


Figure 11-3. *A simple donut chart*

But, we choose to use a donut chart instead of a pie chart because it allows us to represent multiple series at the same time and thus to compare the proportions of its components. Therefore, to continue with the example, you can compare the food consumed by two different groups of people. Listing 11-7 illustrates how to add another data array.

Listing 11-7. ch11_02.html

```
var data2 = [
    ['Dairy', 185],['Meat', 166], ['Grains', 243],
    ['Fish', 166],['Vegetables', 499], ['Fruit', 370]
];
```

Adding this second array to `data`, you modify the listing:

```
$.jqplot ('myChart', [data, data2], options);
```

Figure 11-4 presents the donut chart reporting the two series of values.

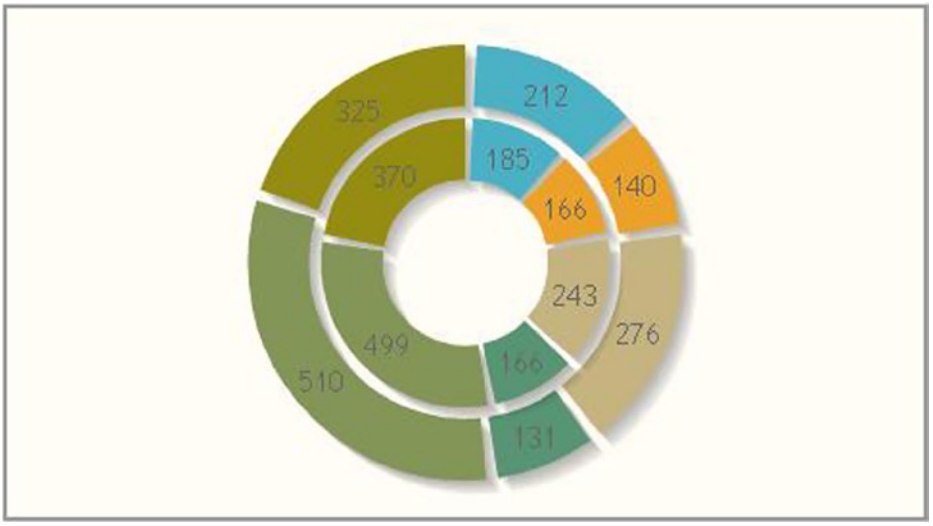


Figure 11-4. *A multiseries donut chart*

Looking at Figure 11-4, you can see at once that something fundamental is missing: a legend. A legend is required because the plug-in automatically assigns a color to each sector, and so without a color reference, it is very hard to understand the chart. Hence, after you set the `show` property of the legend to `'true'`, you can select the location of the legend. To determine in which position to place the legend, jqPlot uses the location property, to which values corresponding to the cardinal directions are assigned: `'n'` (north), `'s'` (south), `'e'` (east), and `'w'` (west). But, it is also possible to use a combination, for instance, `'ne'`, to indicate the northeast position.

Let us say you decide to locate the legend on the right side of the chart, so you assign `'e'` to the `location` property (see Listing 11-8).

Listing 11-8. ch11_02.html

```
legend: {
    show:true,
    location: 'e'
}
```

The legend automatically reports the labels contained in the data arrays, as displayed in Figure 11-5.

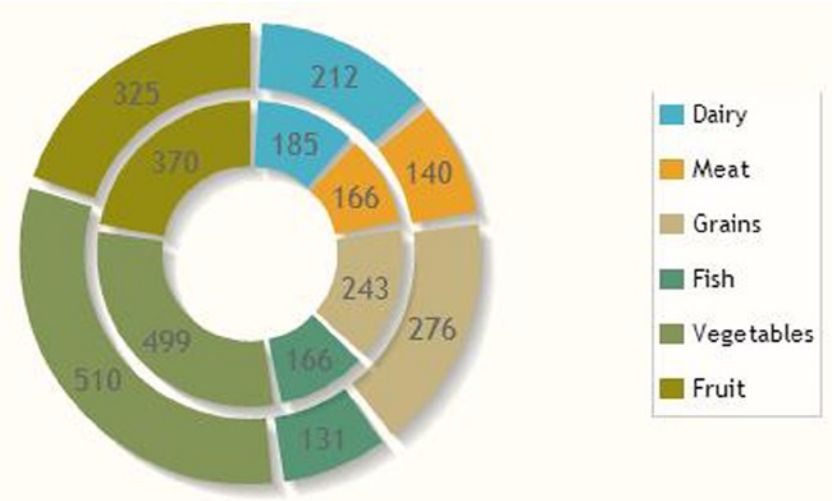


Figure 11-5. *A multiseries donut chart with a legend*

Multilevel Pie Charts

The multilevel pie chart is a modern format that is good for visualizing data that are used for displaying hierarchical relationships. This kind of chart offers a hierarchical structure, starting from a root node in the center of the circle, and you can follow the memberships as they gradually move into the outer circles. To better understand this kind of chart, let us take as an example a series of animals and gradually determine their hierarchical groups.

As input data array, you want to insert three arrays (see Listing 11-9). This will generate three levels of hierarchy. In the first array, you insert the last level, up to the third array, which represents the root.

Listing 11-9. ch11_03.html

```
var data = [ ['Cat', 1],['Dog', 1], ['Mouse', 1],['Snake', 1],
    ['Turtle', 1], ['Jellyfish', 1], ['Cuttlefish', 1] ];
var data2 = [ ['Mammals', 3],['Reptiles', 2], ['Mollusks', 2] ];
var data3 = [ ['Vertebrates', 5],['Invertebrates', 2] ];
```

To generate a multilevel pie chart, you actually need to modify a donut chart, setting the diameter of the inner hole to zero. Instead of displaying a numerical value, in this case you need to show the name of the animal or animal group represented by the label; you must set the `dataLabels` property to `'label'`. The last thing to modify is the set of colors. The default colors provided by jqPlot are not adequate, and it is necessary to define a set of colors for each level of the hierarchy.It is preferable to assign similar colors to animals belonging to the same group and to do likewise for the successive levels of the hierarchy. In Listing 11-10, special attention is paid to the sequence of colors that are assigned to each series (hierarchical level).

Listing 11-10. ch11_03.html

```
var options = {
    seriesDefaults: {
       renderer:$.jqplot.DonutRenderer,
```

```
        rendererOptions: {
           showDataLabels: true,
           dataLabels: 'label',
           startAngle: -90,
           innerDiameter: 0,
           ringMargin: 2,
           shadow: false
        }
    },
    series: [
    {
       seriesColors: ['#4bb2c5', '#4baacc', '#4b88aa', '#bbb2c5',
                      '#bbaa99', '#c5dd99', '#dddd77']
    },
    {
       seriesColors:  ['#4bbbbb', '#ccb2c5', '#c5ff99']
    },
    {
       seriesColors:  ['#aa5555', '#a3ffaa']
    }]
};

$.jqplot ('myChart', [data, data2, data3], options);
```

In the end, your efforts are rewarded with the multilevel pie chart in Figure 11-6.

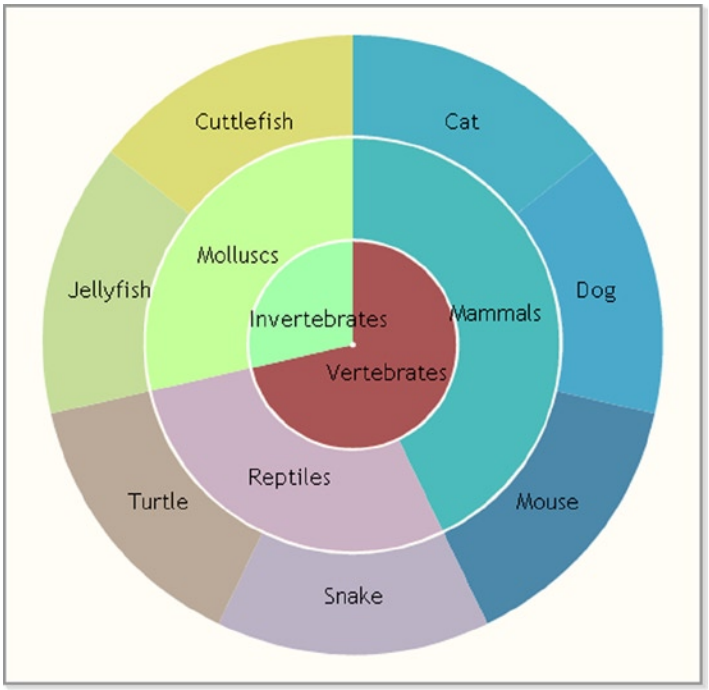


Figure 11-6. *A multilevel pie chart*

Summary

In this chapter, you have seen how the jqPlot library allows you to represent your data by means of a **pie chart** (with a single series of data) or with a **donut chart** (with multiples series of data) while also obtaining a quick overview of some of the main properties and how to set them in options. In the last part of the chapter, you created a **multilevel pie chart**: a classic example of how you can generate a type of chart that is not among the standard charts proposed by the library by modifying certain properties appropriately.

In the next chapter, you will see how the jqPlot library lets you realize **candlestick charts** and how to handle the particular data format **open-high-low-close (OHLC),** which is the basis of this kind of chart.

CHAPTER 12

Candlestick Charts with jqPlot

Candlestick charts are widely used in the analysis of a currency over time or of price movements. This chart consists of a series of vertical bars, called **candlesticks**. They show the opening, closing, lowest, and highest price in a given time period (see Figure 12-1). For this reason, this kind of chart is often called an **OHLC chart** (when it reports open-high-low-close values) or an **HLC chart** (when it reports onlyhigh-low-close values).

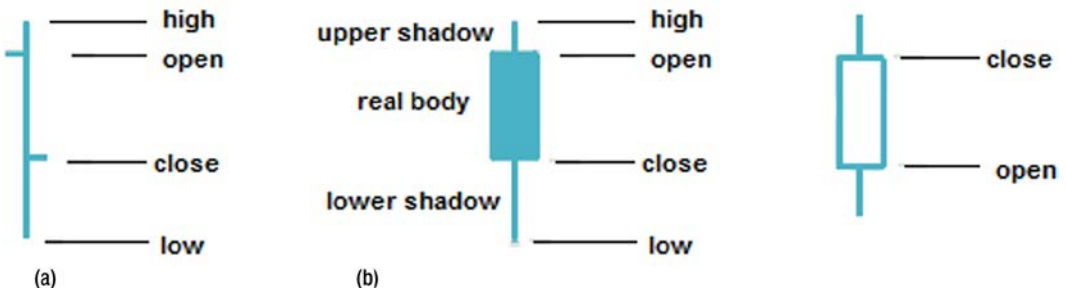


Figure 12-1. *Different ways to represent OHLC data: (a) line, (b) real body*

Candlesticks may be depicted as simple lines or as boxes (called real body) with lines at the ends (called wicks or shadows). The height of each candlestick indicates the price range for a given period. In a box representation the real body is the area between the opening and closing price.If, however, the candlestick is represented by a simple vertical line, two small horizontal ticks indicate the opening (tick to the left) and closing (tick to the right) price. In addition, in candlestick charts, data plots are colored differently, according to whether prices rise or fall.

In this chapter, you will see how particular `OHLC` data can be represented. You will also learn how to format such charts with either lines or real bodies. First, though, you need to include the *OHLCRenderer* plug-in.

OHLC Charts

To enable jqPlot to draw candlestick charts, you must include a specific plug-in in your web page: *OHLCRenderer*.

You need to include the *DateAxisRenderer* plug-in as well, because in candlestick charts, you usually place date values on the x axis:

```
<script type="text/javascript" src="../src/plugins/jqplot.dateAxisRenderer.min.js">
</script>
<script type="text/javascript" src="../src/plugins/jqplot.ohlcRenderer.min.js">
</script>
```

In regard to the input data array, you have to respect a specific order:

```
['timestamp', open, max, min, close]
```

For this example, you are using a set of real data available online. The data are taken from a comma-separated values (CSV) file generated by a free tool, called Dukascopy, which is also available online (www.dukascopy.com). You choose euro–US dollar exchange values from a period of approximately one month in 2012. Let us assign all these values to a variable, as in Listing 12-1.

Listing 12-1. ch12_01a.html

```
var ohlc = [
['8/08/2012 0:00:01', 1.238485, 1.2327, 1.240245, 1.23721],
['8/09/2012 0:00:01', 1.23721, 1.22671, 1.23873, 1.229295],
['8/10/2012 0:00:01', 1.2293, 1.22417, 1.23168, 1.228975],
['8/12/2012 0:00:01', 1.229075, 1.22747, 1.22921, 1.22747],
['8/13/2012 0:00:01', 1.227505, 1.22608, 1.23737, 1.23262],
['8/14/2012 0:00:01', 1.23262, 1.23167, 1.238555, 1.232385],
['8/15/2012 0:00:01', 1.232385, 1.22641, 1.234355, 1.228865],
['8/16/2012 0:00:01', 1.22887, 1.225625, 1.237305, 1.23573],
['8/17/2012 0:00:01', 1.23574, 1.22891, 1.23824, 1.2333],
['8/19/2012 0:00:01', 1.23522, 1.23291, 1.235275, 1.23323],
['8/20/2012 0:00:01', 1.233215, 1.22954, 1.236885, 1.2351],
['8/21/2012 0:00:01', 1.23513, 1.23465, 1.248785, 1.247655],
['8/22/2012 0:00:01', 1.247655, 1.24315, 1.254415, 1.25338],
['8/23/2012 0:00:01', 1.25339, 1.252465, 1.258965, 1.255995],
['8/24/2012 0:00:01', 1.255995, 1.248175, 1.256665, 1.2512],
['8/26/2012 0:00:01', 1.25133, 1.25042, 1.252415, 1.25054],
['8/27/2012 0:00:01', 1.25058, 1.249025, 1.25356, 1.25012],
['8/28/2012 0:00:01', 1.250115, 1.24656, 1.257695, 1.2571],
['8/29/2012 0:00:01', 1.25709, 1.251895, 1.25736, 1.253065],
['8/30/2012 0:00:01', 1.253075, 1.248785, 1.25639, 1.25097],
['8/31/2012 0:00:01', 1.25096, 1.249375, 1.263785, 1.25795],
['9/02/2012 0:00:01', 1.257195, 1.256845, 1.258705, 1.257355],
['9/03/2012 0:00:01', 1.25734, 1.25604, 1.261095, 1.258635],
['9/04/2012 0:00:01', 1.25865, 1.25264, 1.262795, 1.25339],
['9/05/2012 0:00:01', 1.2534, 1.250195, 1.26245, 1.26005],
['9/06/2012 0:00:01', 1.26006, 1.256165, 1.26513, 1.26309],
['9/07/2012 0:00:01', 1.26309, 1.262655, 1.281765, 1.281625],
['9/09/2012 0:00:01', 1.28096, 1.27915, 1.281295, 1.279565],
['9/10/2012 0:00:01', 1.27957, 1.27552, 1.28036, 1.27617],
['9/11/2012 0:00:01', 1.27617, 1.2759, 1.28712, 1.28515],
['9/12/2012 0:00:01', 1.28516, 1.281625, 1.29368, 1.290235] ];
```

In `options` you activate the *OHLCRenderer* plug-in by calling it on the `series` object. Because you need to handle date values on the x axis, you must activate the `dateAxisRenderer` object in the `xaxis` object. With this type of chart, it is better to define the period of time you want to represent, regardless of the input data, in order to have more precise control over what is displayed. To this end, you specify the `min` and `max` properties in `xaxis` object. You can also see that with `dateAxisRenderer`, you can choose the tick interval, using literal expressions (`'1 day'`, `'n days'`, `'1 week'`, `'n weeks'`, `'1 month'`, `'n months'`, where `n` is any integer greater than 1). Furthermore, note that `yaxis` has not been defined or rather that y values have been attributed to `y2axis`. This has been done in order that the y axis be situated on the right edge of the chart rather than the default left edge (see Listing 12-2).

Listing 12-2. ch12_01a.html

```
var options = {
    title: 'EUR-USD Exchange',
    seriesDefaults:{ yaxis: 'y2axis'},
    axes: {
       xaxis: {
          renderer: $.jqplot.DateAxisRenderer,
          tickOptions: {formatString: '%b %e'},
          min: "08-07-2012 16:00",
          max: "09-12-2012 16:00",
          tickInterval: "1 weeks"
       },
       y2axis: {
          tickOptions:{ formatString: '$%.2f'}
       }
    },
    series: [{ renderer: $.jqplot.OHLCRenderer}]
};

$.jqplot('myChart', [ohlc], options);
```

You now have the OHLC chart shown in Figure 12-2.

Figure 12-2. *An OHLC chart with lines*

Insofar as you had integers, you represented them just as they are entered in the input data array. But, this is not always possible. Often, you must deal with numbers that have many digits after the decimal point and that are not of the same length. It is therefore necessary to standardize these numbers, reporting only the significant digits. You can accomplish this by setting the `formatString` property. This particular case requires a float value with two decimal points: `'%.2f'`.

Using Real Bodies and Shadows

The candlestick chart you have just seen is formatted with bar lines. If you want a box representation, with real bodies and shadows, you need to set an additional property: the `candlestick` (see Listing 12-3).

Listing 12-3. ch12_01b.html

```
series: [{
    renderer: $.jqplot.OHLCRenderer,
    rendererOptions:{ candleStick: true }
}]
```

Now, let us look at the real bodies that replace the horizontal ticks on bar lines. In Figure 12-3 the white boxes indicate when the price rises (the opening price is lower than the closing price), black boxes, when the price falls (the closing price is lower than the opening price).

Figure 12-3. *An OHLC chart with boxes*

Comparing Candlesticks

Ocassionally, you will need to compare candlesticks representing different categories at particular time. In such cases, you do not have dates on the x axis, but the names of the subjects themselves. The input data array will be different; you must separate the OHLC data, using the labels of the categories from which they were taken. For each entity, you insert an array with five values:

```
[n, open, max, min, close]
```

Here, instead of the timestamp, `n` is an integer corresponding to the index of the tick array. Thus, you define these OHLC values in the `data1` array, as shown in Listing 12-4. In the `ticks` array, you use four different labels to indicate each of the four OHLC values.

Listing 12-4. ch12_02.html

```
var data1 = [[1, 75, 80, 40, 55], [2, 30, 60, 15, 50],
             [3, 64, 75, 48, 50], [4, 67, 78, 20, 36]];
var ticks = ['Apple', 'Ubuntu', 'Microsoft', 'Android'];
```

You replace the call to the *DateAxisRenderer* plug-in with one to the *CategoryAxisRenderer* plug-in:

```
<script type="text/javascript" src="../src/plugins/jqplot.categoryAxisRenderer.min.js">
</script>
<script type="text/javascript" src="../src/plugins/jqplot.ohlcRenderer.min.js">
</script>
```

As you can see in Listing 12-5, the settings in `options` are very simple. First, you have to replace `$.jqplot.DateAxisRenderer` with `$.jqplot.CategoryAxisRenderer` in the `renderer` property. Furthermore, you assign the `ticks` array to the `ticks` object in the `xaxis`.

Listing 12-5. ch12_02.html

```
var options = {
    axes: {
       xaxis: {
          renderer: $.jqplot.CategoryAxisRenderer,
          ticks: ticks
       },
    },
    series: [{
       renderer: $.jqplot.OHLCRenderer,
       rendererOptions:{ candleStick: true}
    }]
};

$.jqplot ('myChart', [data1], options);
```

This chart gives a perfect picture of a box representation with real bodies; the box is filled when the price falls and empty when the price rises (see Figure 12-4).

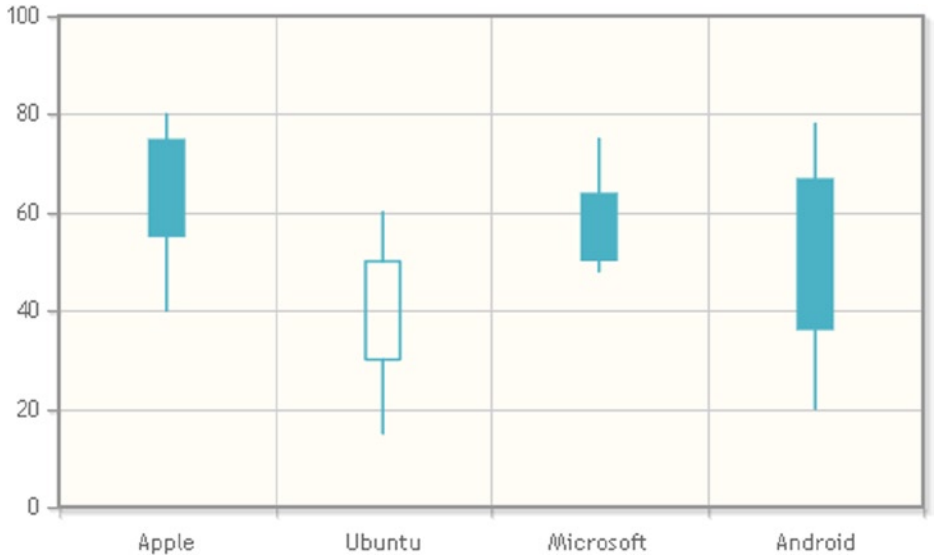


Figure 12-4. *A comparative candlestick chart*

Summary

In this chapter, you have seen how, by means of a **candlestick chart,** particular **OHLC** data can be represented. You have also learned how to format such charts with either lines or real bodies.

In the next chapter, I will discuss a whole class of charts sharing a common feature: their aim is to represent a **distribution** of data. Through this exploration, you will discover how to realize **scatter charts**, **bubble charts**, and **block plots** with the jqPlot library.

CHAPTER 13

Scatter Charts and Bubble Charts with jqPlot

In this chapter, I will discuss a category of charts that are particularly useful when representing a data distribution. It is likely you will often find yourself interested in how a set of data is distributed along the space defined by two different parameters, shown along the x axis and y axis. Such data distribution can suggest correlation or clustering.

The scatter chart is the best choice for the display of data distribution, especially when a large set of data needs to be analyzed. Thus, you will first learn how to realize this kind of chart, using a simple example. Subsequently, you will see how, once two different data groups (clusters) are defined, it is possible to highlight correlations between the x and y variables via trend lines.

Finally, you will analyze two other types of charts: bubble charts and block charts. These may be considered variations of the scatter chart—variations in which data points are replaced with bubbles or blocks. Bubble charts are used when you need to represent data with three different parameters (the scatter chart works only with two); the third parameter is represented by the radius of the bubble. The block chart, is a particular kind of scatter chart, in which, instead of data points, you use a box containing a label.

Scatter Chart (xy Chart)

At first glance, you might think that a scatter chart (also called a scatter plot or xy chart) is a line chart in which the points are not connected, but this would be a mistake. In fact, scatter charts, along with bubble charts and block charts, are a particular type of chart. In a scatter chart, points are represented by the (x, y) pair, but you can get many points with the same x value, making it both difficult and unnecessary to join them with a line. The purpose of a line chart is to follow the progress of a y value in the range of an x value. The purpose of a scatter chart is to display a collection of points that may or may not have some sort of relationship (which can be nonlinear). Furthermore, you may want to analyze these points and their distribution in an (x, y) space, as when, for example, they are distributed in spatially separate groups.

You use the default settings (as in a line chart), disabling the line between the points. Let us take, for instance, two collections of (x, y) data that may present some form of relationship, as shown in Listing 13-1.

Listing 13-1. ch13_04a.html

```
var data = [[400, 35], [402, 37], [650, 55], [653, 56], [650, 50],
            [700, 55], [600, 37], [601, 43], [450, 38], [473, 37],
            [480, 42], [417, 37], [510, 41], [553, 44], [570, 39],
            [527, 41], [617, 41], [625, 49]];
```

```
var data2 = [[100, 40], [600, 80], [200, 50], [300, 55], [400, 60],
             [500, 70], [123, 43], [110, 41], [157, 45], [160, 48],
             [237, 49], [248, 55], [287, 50], [321, 59], [359, 52],
             [387, 62], [466, 68], [533, 74], [344, 60], [323, 51],
             [430, 65]];
```

The points that you have entered do not follow any order, unlike those in a line chart. As previously stated, you use the default settings, disabling the lines between the points by setting the `showLine` property to '`false`' (see Listing 13-2).

Listing 13-2. ch13_04a.html (Disabling the lines between the points using '`false`')

```
var options = {
    title: 'Scatter Chart',
    seriesDefaults: {
       showLine: false,
       showMarkers: true
    }
};
$.jqplot('myChart', [data, data2], options);
```

You thus obtain the scatter chart in Figure 13-1, in which the two collections of data cover two different areas of the chart. The points are divided into well-defined groups.

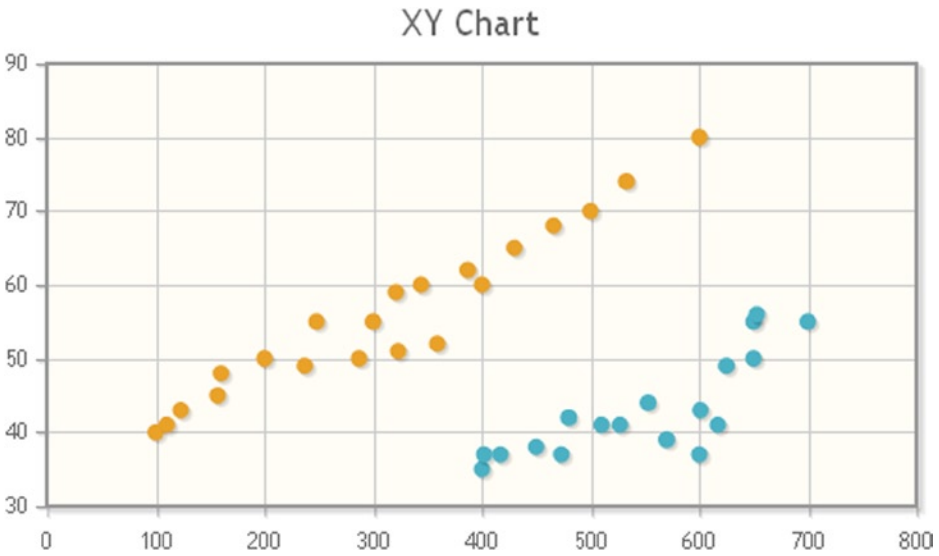


Figure 13-1. *A scatter chart*

Only once the two data collections are represented does it makes sense to determine whether they follow a linear or exponential trend. Here, you can use the trend line functionality of jqPlot. You therefore include the *Trendline* plug-in:

```
<script type="text/javascript" src="../src/plugins/jqplot.trendline.min.js"></script>
```

Or, if you prefer to use a content delivery network (CDN) service, you may do so as follows:

```
<script type="text/javascript"
src="http://cdn.jsdelivr.net/jqplot/1.0.8/plugins/jqplot.trendline.min.js"></script>
```

Then, activate the *Trendline* plug-in for both series, assigning each line a different color (see Listing 13-3).

Listing 13-3. ch13_04b.html

```
var options = {
    title: 'Scatter Chart',
    seriesDefaults: {
       showLine: false,
       showMarkers: true
    },
    series: [{
       trendline: {
          show: true,
          color: '#0000ff',
          type: 'exponential'
       }
    },{
       trendline: {
          show:true,
          color: '#ff0000'
       }
    }]
};
```

As a result, you get a scatter chart with two different series, each with its own trend line, as illustrated in Figure 13-2.

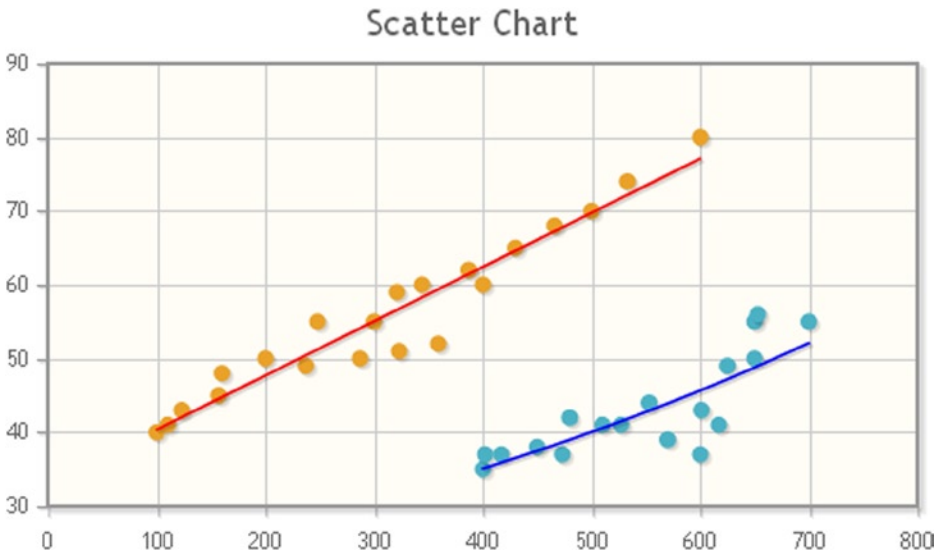


Figure 13-2. *A scatter chart with trend lines*

Bubble Chart

You use a bubble chart when you need to display data in three dimensions. Each entity is therefore represented by a triplet (v1, v2, v3) of independent values. Two of these values are expressed by plotting a disk with an (x, y) point as center. The third value is expressed by the disk radius (r). Hence, the (v1, v2, v3) triplet must be converted to (x, y, r). Which of the three (v1, v2, v3) values is the radius and which is x or y depends on the skill of the chart designer.

Similar to xy charts, bubble charts are often used to identify probable relationships between the data represented or even to see whether they fall into different groups. Such an approach is commonly found in scientific, medical, and economic data analysis.

There is a specific plug-in for bubble charts in jqPlot: *BubbleRenderer*. It is therefore necessary to include this plug-in in your web page:

```
<script type="text/javascript" src="../src/plugins/jqplot.bubbleRenderer.min.js"></script>
```

Or, if you prefer to use a CDN service, you may do so as follows:

```
<script type="text/javascript"
src="http://cdn.jsdelivr.net/jqplot/1.0.8/plugins/jqplot.bubbleRenderer.min.js"></script>
```

The input data array has four values per item:

```
[x, y, radius, <label or object>]
```

The first two values represent the (x, y) coordinates, the third value (pay attention!) is proportional to the radius of the bubble, and the last value represents the reference label (you can actually pass an object, too; more on this in a moment). Listing 13-4 defines an array containing characteristic values of seven European nations: as a value for x, you will insert the surface area; for y, the population; and the radius will represent an economic value. The fourth value is a label reporting the name of the state.

Listing 13-4. ch13_01a.html

```
var data = [[301,60,29392,"Italy"], [675,65,34205,"France"],
            [506,46,30625,"Spain"], [357,81,37896,"Germany"],
            [450,9,37333,"Sweden"], [30,11,37736,"Belgium"],
            [132,11,27624,"Greece"]];
```

Now, let us analyze how to set the `options` variable (see Listing 13-5). You need to activate the *BubbleRenderer* plug-in in the `seriesDefault` object and set the `bubbleGradients` property to 'true'. This will fill the "bubbles" with a color gradient, giving a sense of depth: the disks are thus made to appear as if they were spheres. As you can see, for this plug-in, you do not need to create an array containing labels for the bubbles and then assign it explicitly to an object in `options`; the labels are automatically read by the same input data array. The settings to be specified in `options` are few and simple.

Listing 13-5. ch13_01a.html (Setting the `options` variable)

```
var options = {
    title: 'Bubble Chart with Gradient Fills',
    seriesDefaults:{
       renderer: $.jqplot.BubbleRenderer,
       rendererOptions: {
          bubbleGradients: true
       },
```

```
        shadow: true
    },
    axes: {
        xaxis: {
            label: "Total area [*1000 km3]"
        },
        yaxis: {
            label: "Population [million]"
        }
    }
};

$.jqplot('myChart', [data], options);
```

Finally, with these few rows of code, you get the wonderful bubble chart shown in Figure 13-3.

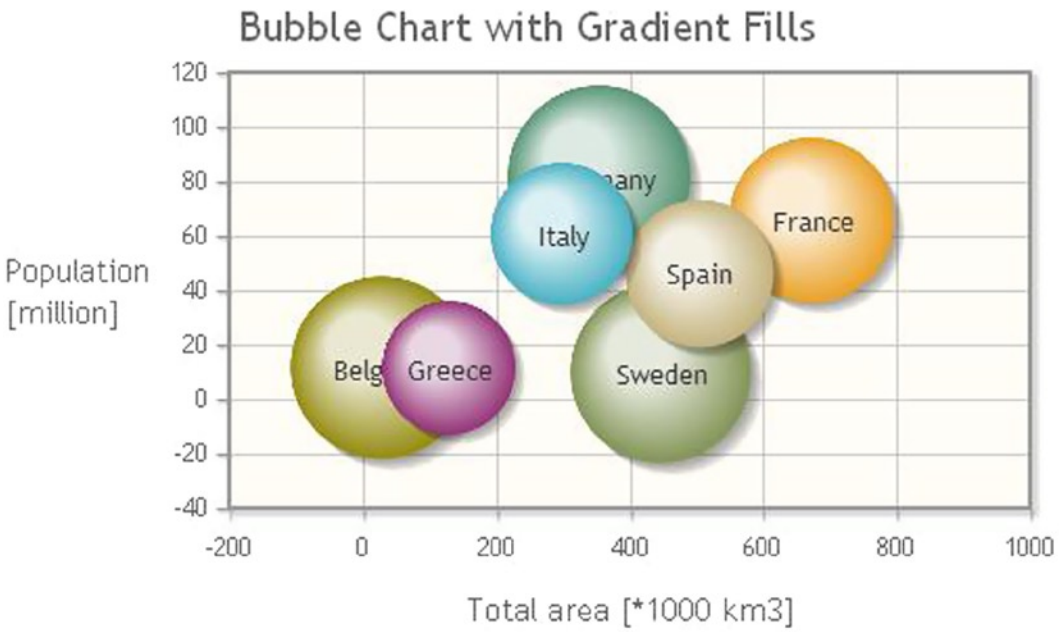


Figure 13-3. *A bubble chart*

Earlier, we I mentioned the possibility of passing an object as the fourth value in the input data array. Here, you can see in detail what this involves. You can pass, simultaneously, an object that allows you to define both the label and the color to be attributed to each individual element (bubble). Using the previous example (see Listing 13-5), you attach colors that are different from those of the default sequence. For example, let us say you want to emphasize the value of one country over that of the others. You take Sweden and assign it the color red. You assign the other countries various shades of brown. You then move the Sweden data to a new array, `data2`; this is to ensure that the "Sweden" bubble is always in the foreground and that it is not overlapped by other bubbles (see Listing 13-6).

Listing 13-6. ch13_02.html

```
var data = [[301, 60, 29392, {label: 'Italy',color:'#b39524'}],
            [675, 65, 34205, {label: 'France', color:'#c39564'}],
            [506, 46, 30625, {label: 'Spain',color:'#a39544'}],
            [357, 81, 37896, {label: 'Germany', color:'#b39524'}],
            [30, 11, 37736, {label: 'Belgium',color:'#c39544'}],
            [132, 11, 27624, {label: 'Greece', color:'#a39564'}]];
var data2 = [[450, 9, 37333, {label: 'Sweden', color:'#ff2524'}]];
```

Using the same `options`, you need only modify the `jqplot()` function, as shown in Listing 13-7. You take this smallshortcut, knowing that the rightmost array item is the one that will be drawn last and that it will, consequently, appear in the foreground.

Listing 13-7. ch13_02.html (modifying the `jqplot()` function)

```
$.jqplot('myChart',[data, data2],options);
```

Figure 13-4 presents the result.

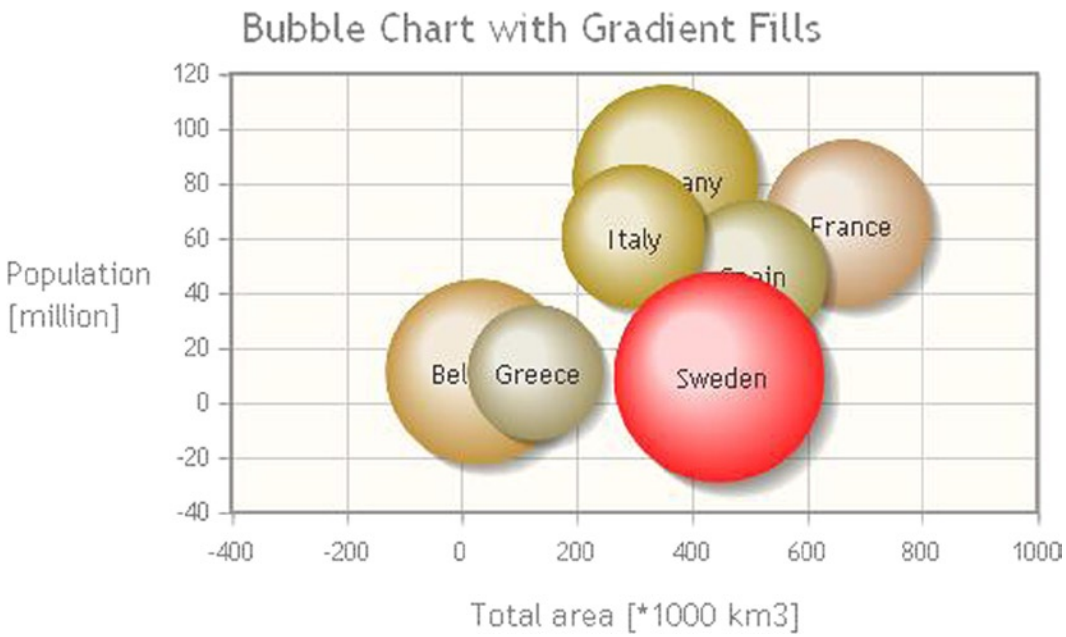


Figure 13-4. *A bubble chart with a selected state in the foreground*

The default sequence of colors suits you fine here, but you decide to change the gradient fill to a transparent effect. To accomplish this, you must add the `bubbleAlpha` property and assign the desired value of transparency to it, as demonstrated in Listing 13-8.

Listing 13-8. ch13_01b.html

```
seriesDefaults:{
    renderer: $.jqplot.BubbleRenderer,
    rendererOptions: {
       bubbleGradients: true,
       bubbleAlpha: 0.6
    },
    shadow: true
},
```

Figure 13-5 shows the bubbles with a gradient fill with a transparent effect that affords a glimpse of the underlying bubbles.

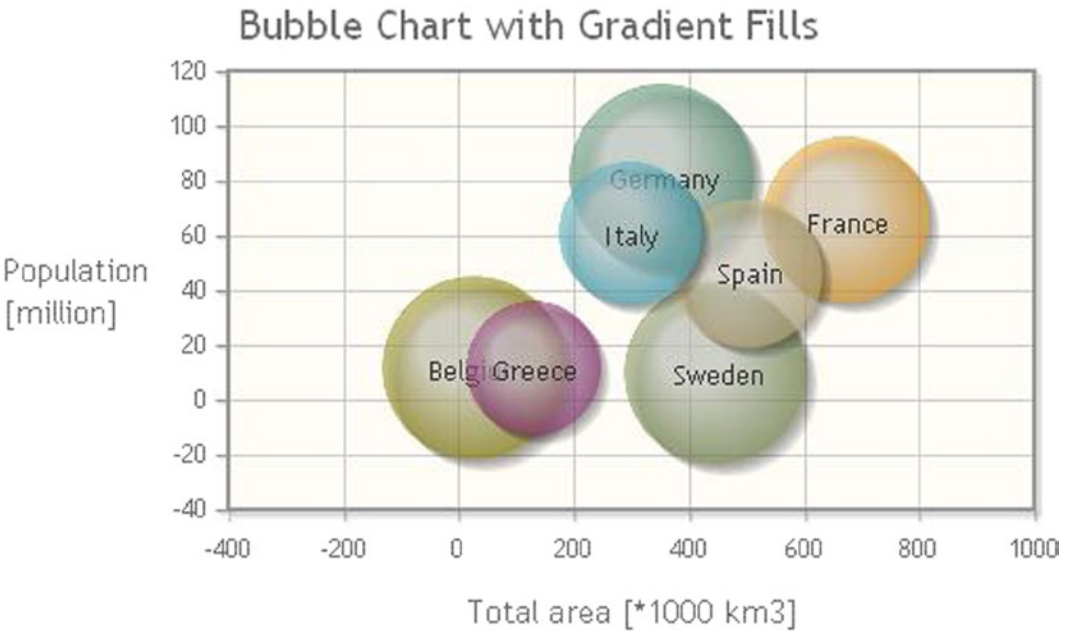


Figure 13-5. *A bubble chart with transparency*

Block Chart

A block chart (also called a block plot) is very similar to the bubble chart, but instead of disks, it uses rectangles. Here, the size of the rectangles has no significance except to provide space for those in which a label is applied to a given (x, y) pair.

As with the bubble chart, it is necessary to include the *BlockRenderer* plug-in in the web page:

```
<script type="text/javascript" src="../src/plugins/jqplot.blockRenderer.min.js"></script>
```

Or, if you prefer to use a CDN service, you may do so as follows:

```
<script type="text/javascript"
src="http://cdn.jsdelivr.net/jqplot/1.0.8/plugins/jqplot.blockRenderer.min.js"></script>
```

In this example you will use three series of data. The input data array should have this format:

```
[x, y, 'Label'],
```

Now, you define three different arrays, as shown in Listing 13-9.

Listing 13-9. ch13_03.html

```
var data1 = [[10, 30, 'Copper'], [100, 40, 'Gold'], [50, 50, 'Silver'],
             [12, 78, 'Lead'], [44, 66, 'Brass']];
var data2 = [[68, 15, 'Maple'], [33, 22, 'Oak'],[10, 90, 'Ebony'],
             [94, 30, 'Beech'],[70, 70, 'Ash']];
var data3 = [[22, 16, 'PVC'], [56, 76, 'PE'], [33, 78, 'PET'],
             [27, 60, 'PC'], [70, 44, 'PU']];
```

In `options`, you need only activate the *BlockRenderer* plug-in in the `seriesDefault` object (see Listing 13-10).

Listing 13-10. ch13_03.html (Activating the `BlockRenderer` plug-in)

```
var options = {
    seriesDefaults:{
       renderer: $.jqplot.BlockRenderer
    }
};

$.jqplot ('myChart', [data1, data2, data3], options);
```

Figure 13-6 gives the block chart you have just defined, in which each series is marked by a different color.

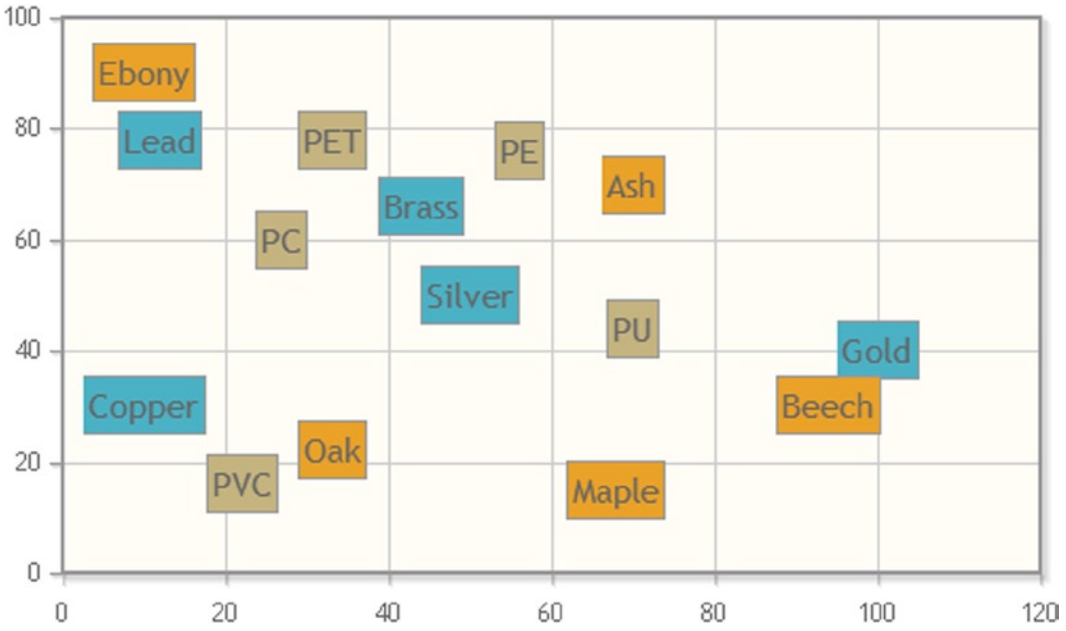


Figure 13-6. *A block chart*

Summary

In this chapter, you have learned how to represent a **distribution**. You will probably often find yourself interested in examining how data are distributed in space in order to uncover any possible trends or clusters. Depending on what you want to stand out more, you can choose to represent data by means of a **scatter chart**, a **bubble chart**, or a **block chart**. In addition, you have seen how to highlight the trend of a distribution.

In the next chapter, I will gather other types of charts that you have not yet looked at but that are standard types belonging to the jqPlot library. First, you will study **funnel charts** and how to set their properties through `options`. Then, you will discover **Bezier curves**—what they are and how they can be implemented by jqPlot.

CHAPTER 14

Funnel Charts with jqPlot

Funnel charts are used to show the progressive reduction of data as they go down one level to the next. The chart consists of an inverted pyramid, or funnel, divided into different levels. Each level has its own area, which is proportional to a given percentage value. A funnel chart is similar to a pie chart in that both express a whole divided into its constituent parts. But, the funnel chart specifies levels, which succeed one another in a very precise sequence. This sequence may express a hierarchical order, the steps of a process, and so on. A pie chart cannot do this.

Creating a Funnel Chart

Even for this specialized chart, jqPlot provides a specific plug-in: *FunnelRenderer*. You therefore need to include it:

```
<script type="text/javascript" src="../src/plugins/jqplot.funnelRenderer.min.js"></script>
```

Or, if you prefer to use a content delivery network (CDN) service, you can do so as follows:

```
<script type="text/javascript"
src="http://cdn.jsdelivr.net/jqplot/1.0.8/plugins/jqplot.funnelRenderer.min.js"></script>
```

With jqPlot, you must be aware of a behavior that is specific to this renderer plug-in: *FunnelRenderer* reorders the data, in descending order. The largest value is displayed at the top of the funnel, with the lesser values placed below. The area of each funnel section corresponds to the value of its data point, relative to the sum of all values (percentage). With this renderer, you need to use the following format for the input data array:

```
['label',value]
```

Thus, for this example, you define the data array as shown:

```
var data = [['Sony', 1], ['Samsung', 13], ['LG', 14], ['Philips', 5]];
```

For `options`, you have to activate the *FunnelRenderer* plug-in in the `seriesDefaults` object, and, optionally, you can add a legend reporting the labels of the series, as presented in Listing 14-1.

Listing 14-1. ch14_01a.html

```
var options = {
    seriesDefaults: {
       renderer: $.jqplot.FunnelRenderer
    },
```

```
    title: {
       text: 'Basic Funnel Chart'
    },
    legend: {
       location: 'e',
       show: true
    }
};

$.jqplot('myChart', [data], options);
```

Figure 14-1 is a basic funnel chart.

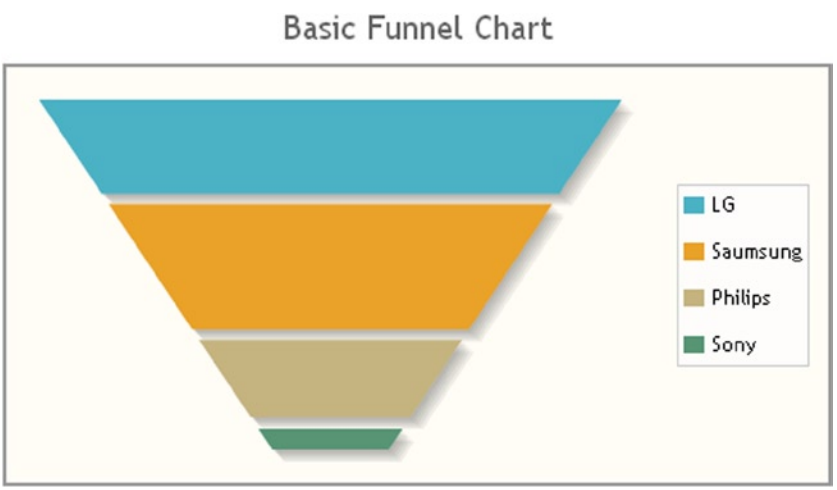


Figure 14-1. *A simple funnel chart*

As you can see, the order of items has been changed, with the element with the highest value at the top, and so on, down to the item with the lowest value. This is the basic chart, but you can enrich it, for example, by adding labels reporting the percentages. To this end, you must add the `dataLabel` property, setting it to '`percent`' and then enabling it with `showDataLabel`, set to '`true`' (see Listing 14-2).

Listing 14-2. ch14_01b.html

```
seriesDefaults: {
    renderer: $.jqplot.FunnelRenderer,
    rendererOptions: {
       dataLabels: 'percent',
       showDataLabels: true
    }
},
```

As you can see in Figure 14-2, percentage values are now reported in their corresponding sections of the funnel chart.

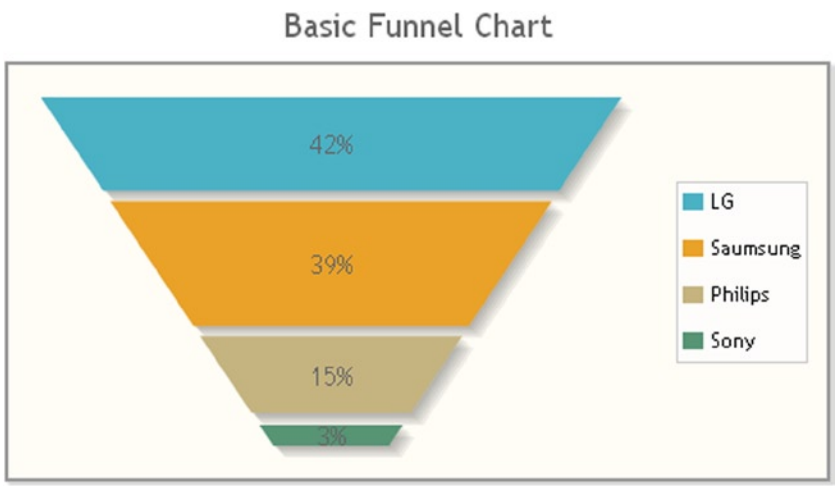


Figure 14-2. *A funnel chart with a legend and percentages*

You can make further changes. For example, let us say you want to decrease the spacing between funnel sections, as in Figure 14-3. You can accomplish this through the values passed to the `sectionMargin` property. By assigning the value 0 to the `sectionMargin` property, you eliminate the space between the sections completely (see Listing 14-3).

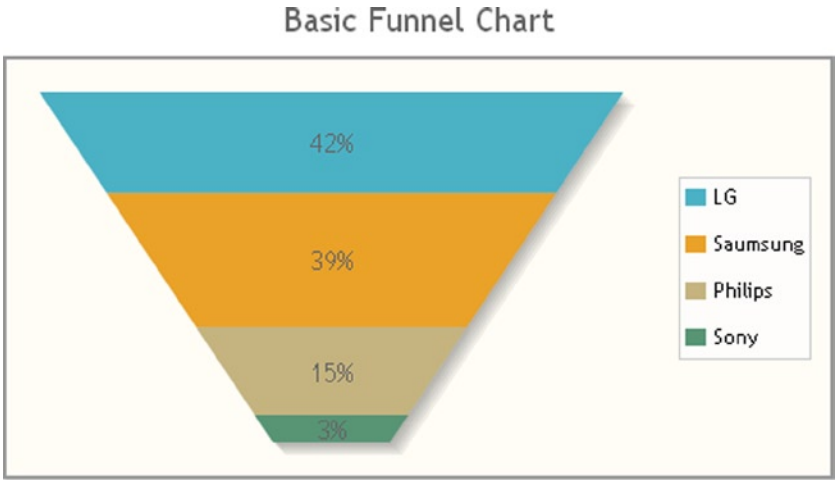


Figure 14-3. *A funnel chart without spaces between sections*

Listing 14-3. ch14_01c.html

```
rendererOptions: {
    dataLabels: 'percent',
    showDataLabels: true,
    sectionMargin: 0
}
```

Alternatively, you may want to represent the various sectors as unfilled and increase the width of their boundary lines, as in Figure 14-4. To do this, you need to use two properties: `fill` and `lineWidth`. First, you set the `fill` property to '`false`' which causes jqPlot to draw the section with an empty area; then, you set the `lineWidth` property to 4, thereby increasing the thickness of the sections' edges, making them more visible (see Listing 14-4).

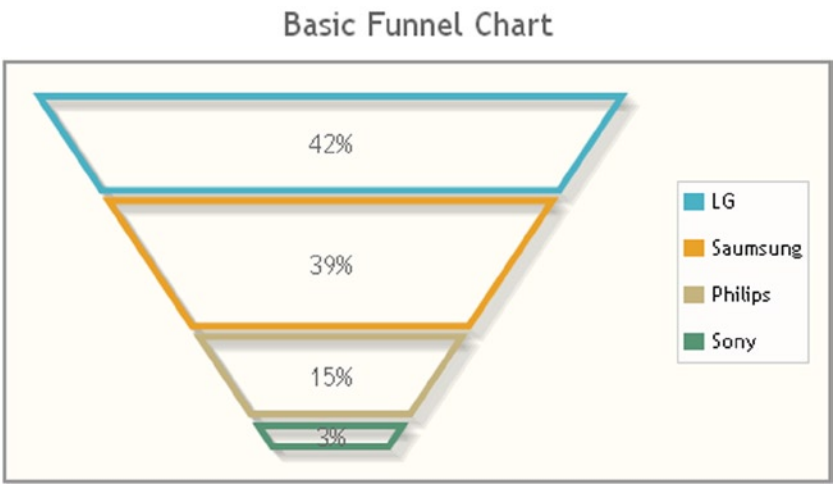


Figure 14-4. *A funnel chart with no filled sections*

Listing 14-4. ch14_01d.html

```
rendererOptions: {
    dataLabels: 'percent',
    showDataLabels: true,
    fill: false,
    lineWidth: 4
}
```

Summary

In this chapter, you learned how to make certain types of **funnel charts** and how to change their properties through `options`.

In the next chapter, I will cover a topic I have touched on in previous chapters: controls. I will describe the importance of introducing controls in a chart, understanding that a property of `options` is hidden behind every control.This affords the user the opportunity to select a property's attributes in real time.

CHAPTER 15

Adding Controls to Charts

Sometimes, it can be useful to change settings directly from the browser at runtime and then replot the chart with these new settings. A typical way of doing this is to add active controls. These controls make the chart interactive, allowing the user to make choices in real time, such as deciding how the chart should be represented.By inserting controls, you give the user the ability to control the values of the chart's attributes, which you would normally have to set in `options`.

In this chapter, you will look at introducing controls within your web page. You will also consider the factors that lead to the choice of one type of control over another. A series of examples featuring three of the most commonly used controls, will take you deeper into this topic.

Adding Controls

One way to group controls is according to their functionality.Some controls (e.g., buttons, menus) work as switches (command controls) with which the user can trigger a particular event or launch a command. Other controls (e.g., check boxes, radio buttons, combo boxes, sliders) are bound to a specific value or property. With this type of control, the user makes a choice or enters values through a text field (text area). Still other controls (e.g., scrollbars) have a navigation function and are especially suitable in situations in which it is necessary to move an object, such as a selected item in a list or a large image enclosed in a frame or in the web page.

Here, you will be investigating those controls that are linked to values and that let the user interact with a chart by making choices. These controls should, in some way, graphically represent the values that a particular property can assume (the same values that you would usually assign to the properties within the `options` object, limited to those that you want to make available to the user). Your choice of control will depend on the property to set and the values that it can assume:

- To enable the user to make a single selection from a set of values (e.g., one of three possible colors), the choice of mutually exclusive **radio buttons** as controls is optimal (see Figure 15-1a).
- To let the user select which series should be visible in a chart, you will need to use **check boxes** (see Figure 15-1b).
- To allow the user to choose within a range of values for a particular attribute (e.g., changing the color of an object through adjustment of the red-green-blue (RGB) values that define the color), a **slider** is generally the best choice (see Figure 15-1c) (in this case, you would use three sliders as controls, corresponding to the colors red, green, and blue).

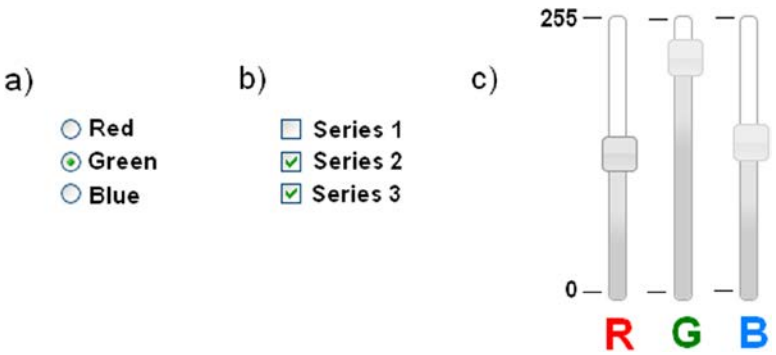


Figure 15-1. *Three of the most commonly used controls: (a) radio buttons, (b) check boxes, (c) sliders*

The list of possible controls does not end there. But, an understanding of the mechanisms that underlie these controls enables a chart developer to handle the vast majority of cases, including those that are the most complex.

In the following examples, you will discover how to apply these three controls to your chart.

Using Radio Buttons

To illustrate the use of controls, let us first look at radio buttons. Radio buttons are a set of small buttons grouped in list form (see Figure 15-1a). They are generally represented as small, empty circles, with text to the side. As previously stated, this type of control is linked to a certain value or property. The particularity of radio buttons is that their values are mutually exclusive; therefore, the user can choose only one of them.

By way of illustration, let us take a simple multiseries line chart, in which, instead of displaying all the series, you want to allow the user to decide which series will be shown. To make a selection, the user will click one of the radio buttons, filling the circle with a dot. The series corresponding to that control will then be drawn on the chart.

Adding Radio Button Controls

First, you need to write the HTML page, importing all the necessary libraries (see Listing 11-1).

Listing 15-1. ch15_01.html

```
<HTML>
<HEAD>
<TITLE>Selection series with controls</TITLE>
<!--[if lt IE 9]>
<script type="text/javascript" src="../src/excanvas.js"></script>
<![endif]-->
<script type="text/javascript" src="../src/jquery.min.js"></script>
<script type="text/javascript" src="../src/jquery.jqplot.min.js"></script>
<link rel="stylesheet" type="text/css" href="../src/jquery.jqplot.min.css" />
<script>
$(document).ready(function(){

    //add your code here

});
```

```
</script>
</HEAD>
<BODY>
<div id="myChart" style="height: 300px; width: 500px;"></div>
    <!-- add the table with the controls here -->
</BODY>
</HTML>
```

Or, if you prefer to use the content delivery network (CDN) service, you use the following code:

```
<!--[if lt IE 9]>
<script src="http://cdn.jsdelivr.net/excanvas/r3/excanvas.js"></script>
<![endif]-->
<script src="http://code.jquery.com/jquery-1.9.1.min.js"></script>
<script type="text/javascript"
    src="http://cdn.jsdelivr.net/jqplot/1.0.8/jquery.jqplot.min.js"></script>
<link rel="stylesheet" type="text/css"
    href="http://cdn.jsdelivr.net/jqplot/1.0.8/jquery.jqplot.min.css" />
```

You start with a line chart in which you will be representing four sets of values. Every element in each series will be represented by an (x, y) pair of values; you insert the values of these four series in a data set defined within the jQuery `$(document).ready()` function, as shown in Listing 15-2.

Listing 15-2. ch15_01.html

```
var dataSet = {
    data1: [[1, 1], [2, 2], [3, 3], [4, 2], [5, 3], [6, 4]],
    data2: [[1, 3], [2, 4], [3, 5], [4, 6], [5, 5], [6, 7]],
    data3: [[1, 5], [2, 6], [3, 8], [4, 9], [5, 7], [6, 9]],
    data4: [[1, 7], [2, 8], [3, 9], [4, 11], [5, 10], [6, 11]]
};
```

But, instead of displaying all four series with lines of different colors, as seen previously, you provide the user the opportunity to display only one series at a time. Once the chart is loaded in the browser, the userwill be able to select any one of the four series and switch between them, without having to load a new page.

You begin by representing only the first series (`data1`) (see Listing 15-3).

Listing 15-3. ch15_01.html

```
var options = {
    seriesDefaults: {
       showMarker: false
    },
    title: 'Series selection',
    axes: {
       xaxis: {},
       yaxis: {
          min: 0,
          max: 12
       }
    }
};

var plot1 = $.jqplot('myChart', [dataSet.data1], options);
```

■ **Note** In this example, you store the value returned by the `$.jqplot()` function within the `plot1` variable. This allows you to access the contents of `jqplot` object, change the values, and call its methods, including the `replot()` function, which lets you draw the chart again, including the new changes.

The user will be selecting an option from a set of possible choices; the radio buttons is the best choice of control for this purpose. Therefore, let us assign one series to each radio button. As you can see in Listing 15-4, all the controls (buttons) are contained in an inner list within a table. Each button is specified by an `<input>` element in which the four series are also specified as values.

Listing 15-4. ch15_01.html

```
<table>
<tr>
  <td>
    <div>
      <ul>
        <li><input name="dataSeries" value="data1" type="radio" checked />First Series</li>
        <li><input name="dataSeries" value="data2" type="radio" />Second Series</li>
        <li><input name="dataSeries" value="data3" type="radio" />Third Series</li>
        <li><input name="dataSeries" value="data4" type="radio" />Fourth Series</li>
       </ul>
    </div> </td>
</tr>
</table>
```

However, setting the controls definition in an HTML page is not enough; you must also create functions that relate the radio buttons to the jqPlot chart. Depending on which radio button is in the `checked` state, a different set from the data set will be loaded in the chart.

In selecting a different radio button, the user changes the checked attribute from `'false'` to `'true'`. The status change of a radio button involves the activation of the `change()` function, which detects this event. This function assigns a new set from the data set to the `plot1` variable (containing all the information about your jqPlot chart) and finally forces the replot of the chart. The new data are thus represented in the chart, without having to reload the page (see Listing 15-5).

Listing 15-5. ch15_01.html

```
$(document).ready(function(){
    ...
    var plot1 = $.jqplot ('myChart', [dataSet.data1], options);
    $("input[type=radio][name=dataSeries]").attr("checked", false);
    $("input[type=radio][name=dataSeries][value=data1]").attr("checked", true);
    $("input[type=radio][name=dataSeries]").change(function(){
       var val = $(this).val();
       plot1.series[0].data = dataSet[val];
       plot1.replot();
    });
});
```

To customize the elements within the table of controls, you can add a little bit of Cascading Style Sheets (CSS) style, as demonstrated in Listing 15-6.

Listing 15-6. ch15_01.html

```
<style>
li {
    font-family: "Verdana";
    font-size: 16px;
    font-weight: bold;
    text-shadow: 1px 2px 2px #555555;
    margin: 3px;
    list-style: none;
}
</style>
```

If you load this web page in the browser, you obtain the chart in Figure 15-2.

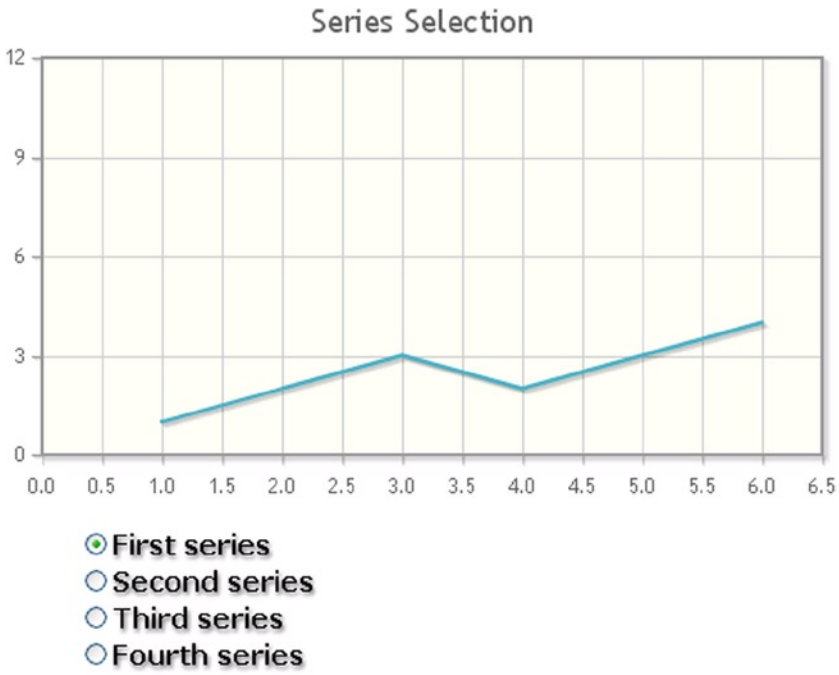


Figure 15-2. *With radio buttons it is possible to select only one series of data*

Now, the user can choose which series will be shown in the chart. Having selected the radio button as a control, the chart will display only one set of data at a time.

Accessing Attributes after the Chart Has Already Been Drawn

So far, you have used the `options` object to define the property values of your chart (by changing the default values) and then passed it as an argument to the `$.jqplot()` function. But, this applies only when you want to characterize your chart before it is drawn. What can you do if you need to access the attribute values subsequently?

In fact, by introducing the controls as an argument, you have also introduced the possibility of changing these attributes after the chart has been drawn. Therefore, there must be a way to access these values, edit them, and then run the command to redraw the chart (as when you used the `replot()` function) (see Listing 15-5).

You have seen that you can receive the entire `jqplot` object as the value returned by the `$.jqplot()` function and store it in a variable (in the previous example, the `plot1` variable) so that you can access its content later.

A `jqplot` object contains practically all the objects—their properties and methods—that define the whole jqPlot library, and every particular instance (e.g., `plot1`) is realized in the representation of a specific chart.

Thus, when you are writing a JavaScript code to define the functions that handle particular events (such as the use of controls by the user), you can access these values and change them after the page has designed the chart and then run the command to redraw it with the desired changes. This adds the interactivity you need in your charts.

Continuing with the previous example (see Listings 15-1 to 15-6), you note that the lines are all drawn in blue. Let us now make some changes so that this time the user can choose the color with which the series will be drawn.

To do this, you add another set of controls to the table: a second column of radio buttons, each representing a color (see Listing 15-7).

Listing 15-7. ch15_02.html

```
<table>
<tr>
  <td>
    <div>
       <ul>
          <li><input name="dataSeries" value="data1" type="radio" checked />First series</li>
          <li><input name="dataSeries" value="data2" type="radio" />Second series</li>
          <li><input name="dataSeries" value="data3" type="radio" />Third series</li>
          <li><input name="dataSeries" value="data4" type="radio" />Fourth series</li>
      </ul>
    </div>
  </td>
  <td>
    <div>
       <ul>
          <li><input name="colors" value="#4bb2c5" type="radio" checked />Blue</li>
          <li><input name="colors" value="#ff3333" type="radio" />Red</li>
          <li><input name="colors" value="#44bb44" type="radio" />Green</li>
          <li><input name="colors" value="#ffaa22" type="radio" />Orange</li>
      </ul>
    </div>
  </td>
</tr>
</table>
```

Next, you add the rows highlighted in bold in Listing 15-8 to your JavaScript code.

Listing 15-8. ch15_02.html

```
$("input[type=radio][name=dataSeries]").attr("checked", false);
$("input[type=radio][name=dataSeries][value=data1]").attr("checked", true);
$("input[type=radio][name=dataSeries]").change(function(){
    var val = $(this).val();
    plot1.series[0].data = dataSet[val];
    plot1.series[0].renderer.shapeRenderer.strokeStyle = col;
    plot1.replot();
});

var col = "#4bb2c5";
$("input[type=radio][name=colors]").change(function(){
    col = $(this).val();
});
```

Figure 15-3 illustrates how the user can decide the series to represent, selecting among four different colors. This is an example of how adding controls increases the interactivity between the user and chart.

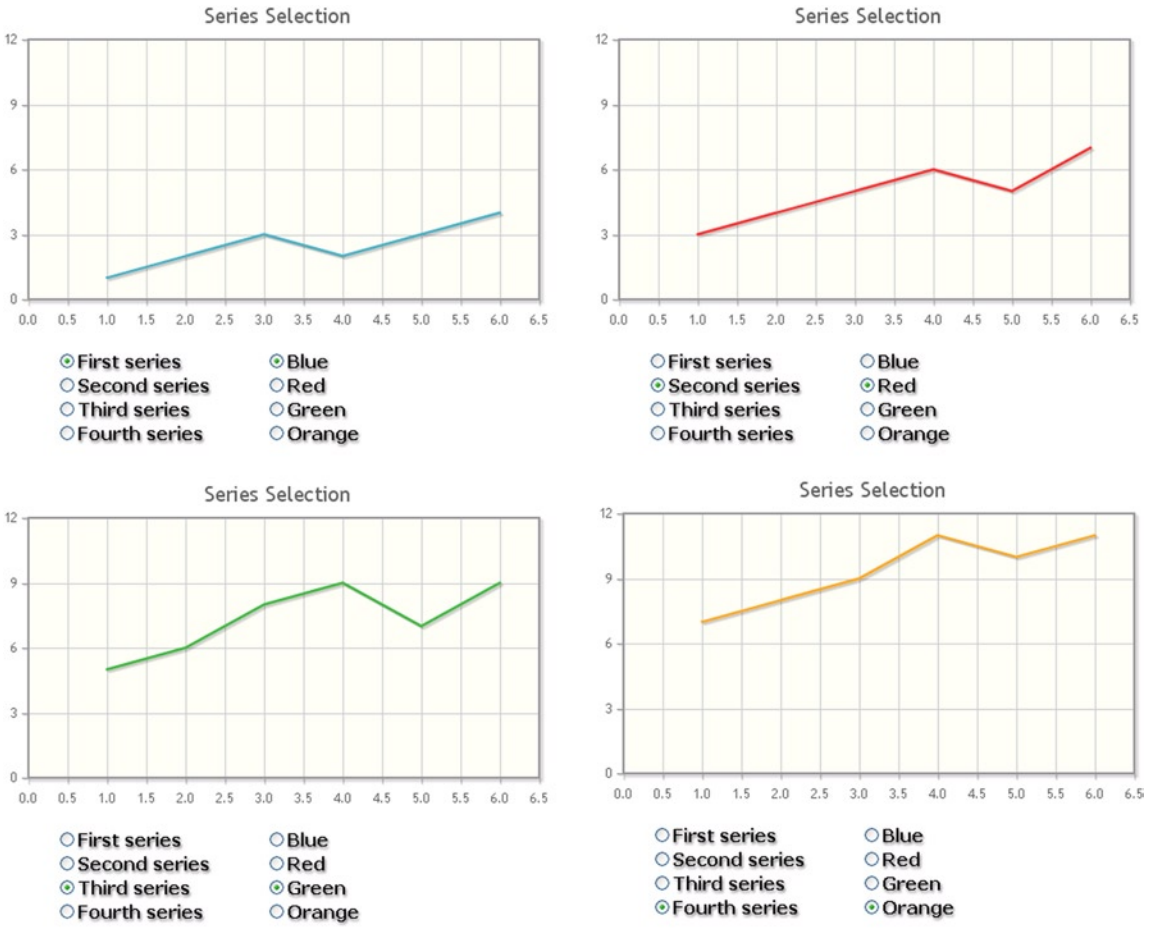


Figure 15-3. *The user can select a different combination of colors and series*

Using Sliders

In the previous example the user first set the color by checking one of the radio buttons in the second column and then chose the series to be represented with that color by selecting it from the first column. This process, therefore, involved two selections, made at two different times. This time, you will keep unchanged the first column, from which the user selects the series to be displayed (mutual exclusion), but in place of the column of radio buttons, you will insert a set of three sliders. In this scenario, the user selects the series to be displayed, and, once it is drawn on the chart, in a predefined color, he or she can modify this color by adjusting the three RGB values that compose it. Now, you have a selection, followed by a fine adjustment.

When you are required to change the value of an attribute by scrolling through contiguous values in a given range, sliders are the kind of control needed. In this case, three sliders are necessary, one for each color (red, green, blue), so that the user can adjust the RGB values to obtain the desired color.

Using the previous example (see Listings 15-7 and 15-8), first you choose the jQuery Interface library (jQuery UI) to obtain the sliders (for details on how to implement the slider using jQuery UI widgets, see Chapter 2). Thus, before adding the sliders to the web page, you must import all the necessary files that are part of this library:

```
<link rel="stylesheet"
    href="http://code.jquery.com/ui/1.10.3/themes/smoothness/jquery-ui.css" />
<script src="http://code.jquery.com/ui/1.10.3/jquery-ui.min.js"></script>
```

■ **Note** If you are working in the workspace made available with the source code that accompanies this book (see Appendix A), you may access the libraries already contained in the workspace by using the following references:

```
<link rel="stylesheet" href="../src/css/smoothness/jquery-ui-1.10.3.custom.min.css" />
<script type="text/javascript" src="../src/js/jquery-ui-1.10.3.custom.min.js"></script>
```

Once you have imported all the files, you can start inserting the three sliders in the HTML table. As you can see in Listing 15-9, you eliminate the second column, containing the radio buttons, replacing it with a set of `<div>` elements (if you are starting directly from here, you can copy the entire listing instead of just the text in bold). The jQuery UI will convert them into sliders (see Chapter 2).

Listing 15-9. ch15_04.html

```
<table>
<tr>
   <td>
      <div>
         <ul>
           <li><input name="dataSeries" value="data1" type="radio" checked />First series</li>
           <li><input name="dataSeries" value="data2" type="radio" />Second series</li>
           <li><input name="dataSeries" value="data3" type="radio" />Third series</li>
           <li><input name="dataSeries" value="data4" type="radio" />Fourth series</li>
         </ul>
      </div>
   </td>
```

```
   <td>
       <div id="red">
          <div id="slider-text">
             <div id="0">0</div>
             <div id="1">255</div>
          </div>
       </div>
       <div id="green">
          <div id="slider-text">
             <div id="0">0</div>
             <div id="1">255</div>
          </div>
       </div>
       <div id="blue">
          <div id="slider-text">
             <div id="0">0</div>
             <div id="1">255</div>
          </div>
       </div>
   </td>
</tr>
</table>
```

Furthermore, you have also added two numerical values to each slider with the `slider-text id`. These values are nothing more than labels that are used to display the minimum and maximum for the range of values (0–255) covered by the three sliders. This methodology can be very useful when you have to represent a scale for each slide in the web page.

Let us now add all the CSS style directives to make sure these new controls can be displayed correctly in the context of the existing page (see Listing 15-10).

Listing 15-10. ch15_04.html

```
<style>
...
#red, #green, #blue {
    float: left;
    margin: 15px;
    left: 50px;
}
#red .ui-slider-range {
    background: #ef2929;
}
#red .ui-slider-handle {
    border-color: #ef2929;
}
#green .ui-slider-range {
    background: #8ae234;
}
#green .ui-slider-handle {
    border-color: #8ae234;
}
```

```
#blue .ui-slider-range {
    background: #729fcf;
}
#blue .ui-slider-handle {
    border-color: #729fcf;
}
#slider-text div {
    font-family: "Verdana";
    font-size: 10px;
    position: relative;
    left: 17px;
}
</style>
```

With regard to the section of code in JavaScript, you keep only the part that manages the radio buttons for the selection of the desired series, integrating it with a new section of code that handles the RGB values, adjusted through the three sliders, as shown in Listing 15-11. The three RGB values are then converted to hexadecimal numbers through an appropriate function and combined to form the HTML color code, expressed by a pound sign (`#`), followed by six hexadecimal characters (`'rrggbb'`), where each pair represents a value from 0 to 255, translated into hexadecimal format.

Listing 15-11. ch15_04.html

```
$(document).ready(function(){

...

    $("input[type=radio][name=dataSeries]").attr("checked", false);
    $("input[type=radio][name=dataSeries][value=data1]").attr("checked", true);
    $("input[type=radio][name=dataSeries]").change(function(){
       var val = $(this).val();
       plot1.series[0].data = dataSets[val];
       plot1.series[0].renderer.shapeRenderer.strokeStyle = "#" + col;
       plot1.replot();
    });

    var col = "4bb2c5";

    function hexFromRGB(r, g, b) {
       var hex = [
          r.toString( 16 ),
          g.toString( 16 ),
          b.toString( 16 )
       ];
       $.each( hex, function( nr, val ) {
          if ( val.length === 1 ) {
             hex[ nr ] = "0" + val;
          }
       });
       return hex.join( "" ).toUpperCase();
   };
```

```
    $( "#red, #green, #blue" ).slider({
       orientation: "vertical",
       range: "min",
       max: 255,
       change: refreshPlot
    });

// set col to default "#4bb2c5";
    $( "#red" ).slider( "value", 255 );
    $( "#green" ).slider( "value", 140 );
    $( "#blue" ).slider( "value", 60 );

    function refreshPlot() {
       var r = $( "#red" ).slider( "value" );
       var g = $( "#green" ).slider( "value" );
       var b = $( "#blue" ).slider( "value" );
       var col = hexFromRGB(r, g, b);
       plot1.series[0].renderer.shapeRenderer.strokeStyle = "#" + col;
       plot1.replot();
    }

    $("[id=0]").css('top','90px');
    $("[id=1]").css('top','-20px');

});
```

The last two lines of code in Listing 15-11 use the jQuery `css()` function to assign a CSS style to a specific selection of HTML elements (see Chapter 2). The selection is made on all elements with `id = 0` and `id = 1`, that is, the `<div>` elements containing the labels for the sliders' scale.You set the CSS top attribute to place each scale label next to the corresponding slider, at a specific height.

In Figure 15-4 the user can decide the series to display and change its by modifying the RBG values through three sliders.

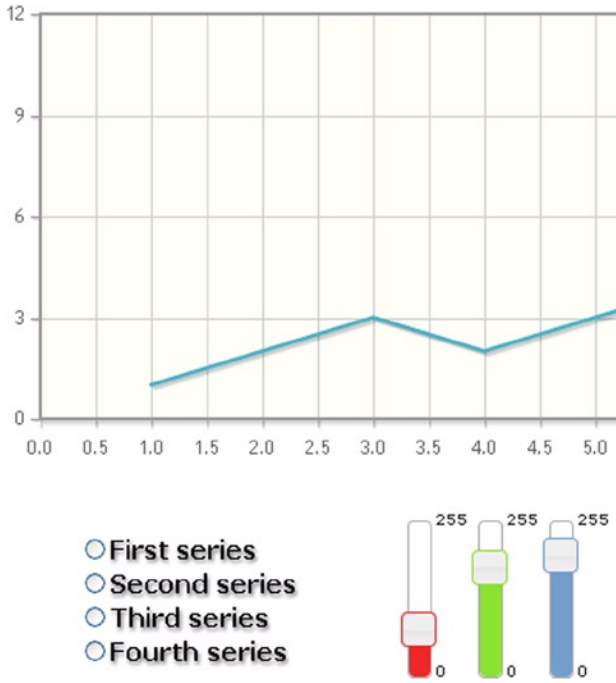


Figure 15-4. *A chart with three slider widgets added to adjust the RGB levels*

Using Check Boxes

In the previous examples, the user could choose only one among the number of series that could be displayed. However, typically the user will want to be able to decide which series should be displayed and which should not, choosing, for instance, to display two or more sets at the same time. This entails dealing with multiple choices within the same group. To enable the user make this kind of choice, you have to opt for check boxes.

Generally, check boxes are grouped in a list, represented by empty boxes (see Figure 15-1). Unlike radio buttons, these controls are not mutually exclusive, but rather multiple choice. Thus, you can select all, some, or none of the values that they represent(whereas with radio buttons an item has to be selected)

Similar to radio buttons, there is a check box for each series, and if a check box is checked, the corresponding series is shown in the chart. Yet, unlike radio buttons, check boxes are independent of each other: their state (checked or unchecked) does not affect the status of the others.

Often, when you have a list of check boxes, it can be very useful to add two buttons with the "CheckAll/UncheckAll" functionality, thereby allowing the choice of selecting/deselecting all the check boxes with one click.

Using the previous example (see Listing 15-9 to 15-11), the data set and options settings are the same; the only thing you need to change is the data passed in the `$.jqplot()` function. In this case, the whole data set will be passed as an argument.

```
var plot1 = $.jqplot ('myChart', [dataSet.data1, dataSet.data2, dataSet.data3, dataSet.data4],
options);
```

Let us delete the table containing the previous controls (radio buttons, sliders) and substitute it with a new one containing check boxes, as shown in Listing 15-12 (if you are starting directly from here, you can copy the entire listing without considering the previous controls). Moreover, in addition to the four controls for as many series, you can add a button at the end to manage the feature "CheckAll/UncheckAll ."

Listing 15-12. ch15_03.html

```
<table>
<tr>
   <td>
      <div>
        <ul>
          <li><input name="data1" type="checkbox" checked />First series</li>
          <li><input name="data2" type="checkbox" checked />Second series</li>
          <li><input name="data3" type="checkbox" checked />Third series</li>
          <li><input name="data4" type="checkbox" checked />Fourth series</li>
          <li><input type="button" name="checkall" value="Uncheck All"></li>
        </ul>
      </div>
    </td>
</tr>
</table>
```

As with radio buttons, you have to add jQuery methods to bind the events that have occurred with these controls. First, you define the status of each check box. Normally, they should all be checked. Then, you define five jQuery methods, enabling or disabling the series to be represented, and then force the replot.

From the code, you must delete all the rows that handled the previous controls and in their place, write the methods in Listing 15-13.

Listing 15-13. ch15_03.html

```
$("input[type=checkbox][name=data1]").change(function(){
    if(this.checked){
       plot1.series[0].data = dataSet.data1;
       plot1.replot();
    } else {
       plot1.series[0].data = [];
       plot1.replot();
    }
});

$("input[type=checkbox][name=data2]").change(function(){
    if(this.checked){
       plot1.series[1].data = dataSet.data2;
       plot1.replot();
    } else {
       plot1.series[1].data = [];
       plot1.replot();
    }
});
```

```
$("input[type=checkbox][name=data3]").change(function(){
    if(this.checked){
       plot1.series[2].data = dataSet.data3;
       plot1.replot();
    } else {
       plot1.series[2].data = [];
       plot1.replot();
    }
});

$("input[type=checkbox][name=data4]").change(function(){
    if(this.checked){
       plot1.series[3].data = dataSet.data4;
       plot1.replot();
    } else {
       plot1.series[3].data = [];
       plot1.replot();
    }
});

$("input[type=button][name=checkall]").click(function(){
    if(this.value == "Check All"){
       plot1.series[0].data = dataSet.data1;
       plot1.series[1].data = dataSet.data2;
       plot1.series[2].data = dataSet.data3;
       plot1.series[3].data = dataSet.data4;
       $("input[type=checkbox][name=data1]").prop("checked", true);
       $("input[type=checkbox][name=data2]").prop("checked", true);
       $("input[type=checkbox][name=data3]").prop("checked", true);
       $("input[type=checkbox][name=data4]").prop("checked", true);
       this.value = "Uncheck All";
       plot1.replot();
    } else {
       plot1.series[0].data = [];
       plot1.series[1].data = [];
       plot1.series[2].data = [];
       plot1.series[3].data = [];
       $("input[type=checkbox][name=data1]").prop("checked", false);
       $("input[type=checkbox][name=data2]").prop("checked", false);
       $("input[type=checkbox][name=data3]").prop("checked", false);
       $("input[type=checkbox][name=data4]").prop("checked", false);
       this.value = "Check All";
       plot1.replot();
    }
});
```

As shown in Figure 15-5, the user can now select the series he or she wants to see displayed in the chart.

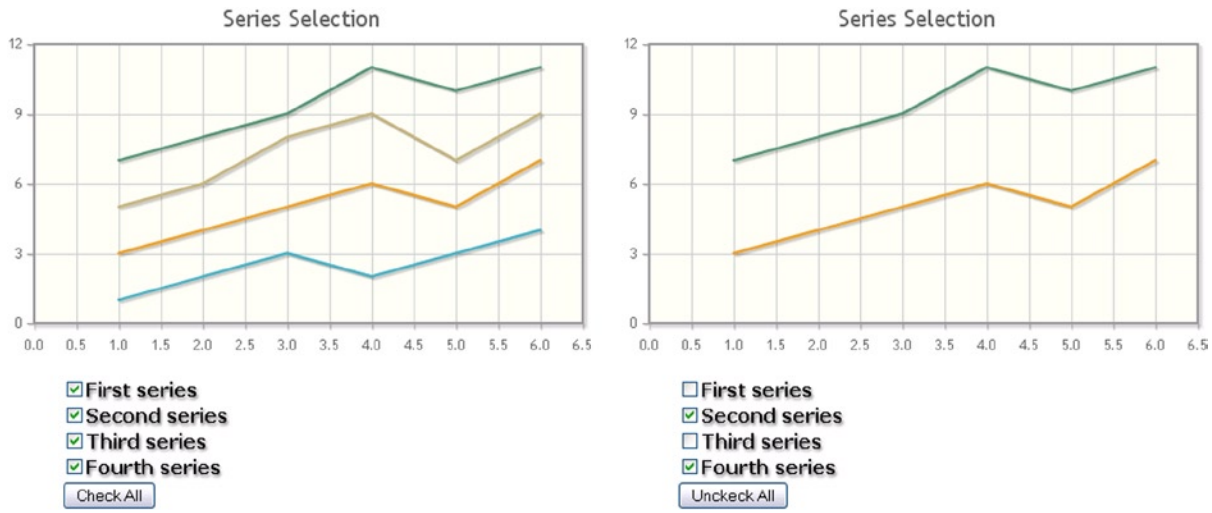


Figure 15-5. *A custom legend with check boxes and a button*

If you click the button labeled "Uncheck all," all the check boxes will be unchecked, and the corresponding series will be hidden in the plot. Subsequently, the button will show the label "Check All." When clicking it this time, all the check boxes will be checked, and the corresponding series will be shown in the chart.

The features covered in this last example are very similar to the legend provided by the *EnhancedLegendRenderer* plug-in (see the section "Handling Legends" in Chapter 10). In that case, by clicking the colored square corresponding to a series, you can decide whether that series should be represented in the chart. But, here you have also added the possibility of checking and unchecking all the series with just one click, and this functionality is not at present implemented in the plug-in (although someone is proposing it). This is another small example of how to expand the functionality that a library provides through the use of controls.

Summary

In this chapter, you have seen how to use various controls, such as **radio buttons**, **sliders**, and **check boxes**, to increase the interactivity of a chart. With the introduction of controls, we, as programmers, are no longer the only ones to have direct control of the values of the properties of the chart; through such controls the user is also able to make the appropriate choices.

In addition, you learned how to integrate **jQuery UI widgets** with the jqPlot library, using these widgets **as controls**. In the next chapter, you will complete this integration by using jQuery UI widgets **as containers** for your charts. This combination greatly expands the possibilities for development and representation of charts using the jqPlot library.

CHAPTER 16

Embedding jqPlot Charts in jQuery Widgets

In Chapter 2, you saw several examples of jQuery UI widgets used as containers. In this chapter, you'll exploit such capability to represent the charts within these containers. This enables you to exploit the great potential of the jQuery UI widgets to further improve the way in which your charts are represented.

The advantages of combining jQuery UI and jqPlot libraries are various: you can display more charts occupying the same space in the web page, and at the same time keep the context of the page clean and tidy. Another advantage is that jQuery UI widgets can be resized, and even users can resize a jqPlot chart.

In this chapter, you'll explore three simple cases where the benefits just mentioned will be made evident. You'll also become more confident in working with jQuery UI widgets, even more so than you did in Chapter 2.

jqPlot Charts on Tabs

The first widget you're going to use as a container is the *tab* (see the section "Tab" in Chapter 2). Inserting charts inside tabs allows you to display different charts on the same page within a limited area. In this example, you'll place three different jqPlot charts within three tabs, called `Tab 1`, `Tab 2`, and `Tab 3`. In the first tab you'll place a bar chart, in the second tab you'll place a multiseries line chart, and in the last tab a pie chart. You won't be analyzing these charts in detail, because they are exactly the same kind of charts used in previous chapters. Each type of chart requires its specific plug-ins, and Listing 16-1 shows a list of the plug-ins that are needed.

Listing 16-1. ch16_01.html

```
<!--[if lt IE 9]>
<script src="http://cdn.jsdelivr.net/excanvas/r3/excanvas.js"></script>
<![endif]-->
<script src="http://code.jquery.com/jquery-1.9.1.min.js"></script>
<script src="http://cdn.jsdelivr.net/jqplot/1.0.8/jquery.jqplot.min.js">
</script>
<link rel="stylesheet" type="text/css"
    href="http://cdn.jsdelivr.net/jqplot/1.0.8/jquery.jqplot.min.css" />
<script src="http://cdn.jsdelivr.net/jqplot/1.0.8/plugins/jqplot.pieRenderer.min.js">
</script>
```

```
<script type="text/javascript" src="http://cdn.jsdelivr.net/jqplot/1.0.8/plugins↩
    /jqplot.dateAxisRenderer.min.js"></script>
<script type="text/javascript" src="http://cdn.jsdelivr.net/jqplot/1.0.8/plugins↩
    /jqplot.canvasTextRenderer.min.js"></script>
<script type="text/javascript" src="http://cdn.jsdelivr.net/jqplot/1.0.8/plugins
    /jqplot.canvasAxisTickRenderer.min.js"></script>
<script type="text/javascript" src="http://cdn.jsdelivr.net/jqplot/1.0.8/plugins
    /jqplot.categoryAxisRenderer.min.js"></script>
<script type="text/javascript" src="http://cdn.jsdelivr.net/jqplot/1.0.8/plugins
    /jqplot.barRenderer.min.js"></script>
```

In addition, you are going to use jQuery widgets as containers, and these also require some files to be included:

```
<link rel="stylesheet" href="http://code.jquery.com/ui/1.10.3/themes/smoothness/jquery-ui.css" />
<script src="http://code.jquery.com/ui/1.10.3/jquery-ui.min.js"></script>
```

■ **Note** For those of you working on the workspace made available with the book's source code, you can use the libraries already contained in the workspace. Use the following references:

```
<script src="../src/js/jquery-1.9.1.js"></script>
<script src="../src/jquery.jqplot.min.js"></script>
<link rel="stylesheet" type="text/css" href="../src/jquery.jqplot.min.css" />

<link rel="stylesheet" href="../src/css/smoothness/jquery-ui-1.10.3.custom.min.css" />
<script src="../src/js/jquery-ui-1.10.3.custom.min.js"></script>
<script src="../src/plugins/jqplot.pieRenderer.min.js"></script>
<script src="../src/plugins/jqplot.dateAxisRenderer.min.js"></script>
<script src="../src/plugins/jqplot.canvasTextRenderer.min.js"></script>
<script src="../src/plugins/jqplot.canvasAxisTickRenderer.min.js"></script>
<script src="../src/plugins/jqplot.categoryAxisRenderer.min.js"></script>
<script src="../src/plugins/jqplot.barRenderer.min.js"></script>
```

With the introduction of so many graphic elements on the web page, the use of Cascading Style Sheets (CSS) styles becomes increasingly important. You need to define some settings in order to modify the tabs' appearance so that they will fit to your needs. Add the style settings in Listing 16-2.

Listing 16-2. ch16_01.html

```
<style>
.ui-tabs {
    width: 690px;
    margin: 2em auto;
}
.ui-tabs-nav {
    font-size: 12px;
}
.ui-tabs-panel {
    font-size: 14px;
}
.jqplot-target {
    font-size: 18px;
}
ol.description {
    list-style-position: inside;
    font-size: 15px;
    margin: 1.5em auto;
    padding: 0 15px;
    width: 600px;
}
</style>
```

You are going to use three different charts, which have already been used in previous chapters (see Chapter 9 for the line chart, Chapter 10 for the bar chart, and Chapter 11 for the pie chart). This chapter, therefore, doesn't cover the details of their settings. Add them to our web page, replacing the usual target name `myChart` with `chart1`, `chart2`, and `chart3`. As you can see in Listing 16-3, in these charts you have defined the `options` objects directly within the three `jqplot()` functions. Their return values are stored in three different variables: `plot1`, `plot2`, and `plot3`. These variables will be used to handle the respective charts within the JavaScript code.

Listing 16-3. ch16_01.html

```
var bar1 = [['Germany', 12], ['Italy', 8], ['Spain', 6],
            ['France', 10], ['UK', 7]];
var data1 = [1, 2, 3, 2, 3, 4];
var data2 = [3, 4, 5, 6, 5, 7];
var data3 = [5, 6, 8, 9, 7, 9];
var data4 = [7, 8, 9, 11, 10, 11];
var pie1 = [
    ['Dairy', 212], ['Meat', 140], ['Grains', 276],
    ['Fish', 131], ['Vegetables', 510], ['Fruit', 325]
];
```

```
var plot1 = $.jqplot ('chart1', [bar1], {
    title: 'Foreigner customers',
    series:[{renderer:$.jqplot.BarRenderer}],
    axesDefaults: {
       tickRenderer: $.jqplot.CanvasAxisTickRenderer ,
       tickOptions: {
          angle: -30,
          fontSize: '10pt'
       }
    },
    axes: {
       xaxis: {
          renderer: $.jqplot.CategoryAxisRenderer
       }
    }
});

var plot2 = $.jqplot ('chart2', [data1, data2, data3, data4],{});

var plot3 = $.jqplot ('chart3', [pie1], {
    seriesDefaults: {
       renderer: jQuery.jqplot.PieRenderer,
       rendererOptions: {
          showDataLabels: true,
          dataLabels: 'value',
          fill: false,
          sliceMargin: 6,
          lineWidth: 5
       }
    }
});
```

Now it is time to add the jQueryUI `tabs()` function at the end of the `$(document).ready()` function, as shown in Listing 16-4.

Listing 16-4. ch16_01.html

```
$(document).ready(function(){
...
    $("#tabs").tabs();
});
```

This call creates the tabs container, and consequently you need to bind the tabs to your plots (see Listing 16-5).

Listing 16-5. ch16_01.html

```
$('#tabs').bind('tabsshow', function(event, ui) {
    if (ui.index === 0 && plot1._drawCount === 0) {
       plot1.replot();
    }
    else if (ui.index === 1 && plot2._drawCount === 0) {
       plot2.replot();
    }
    else if (ui.index === 2 && plot3._drawCount === 0) {
       plot3.replot();
    }
});
```

Selecting a tab will replot the content of the chart within it. Now, in the `<body>` part of the web page, you need to add the `<div>` elements that the jQuery UI library will convert into tabs. The way to do that is to specify a `<div>` element with `tabs` as `id`. Inside it, you define a list of three items, each representing a tab. After the list, you must define another three subdivisions of `tabs`: three additional `<div>` elements called `tabs-1`, `tabs-2`, and `tabs-3`. You are going to put these into your charts: `chart1`, `chart2`, and `chart3` (see Listing 16-6).

Listing 16-6. ch16_01.html

```
<div id="tabs">
    <ul>
       <li><a href="#tabs-1">Tab 1</a></li>
       <li><a href="#tabs-2">Tab 2</a></li>
       <li><a href="#tabs-3">Tab 3</a></li>
    </ul>
    <div id="tabs-1">
       <p>This is the bar chart</p>
       <div id="chart1" style="height:300px; width:650px;"></div>
    </div>
    <div id="tabs-2">
       <p>This is the line chart</p>
       <div id="chart2" style="height:300px; width:650px;"></div>
    </div>
    <div id="tabs-3">
       <p>This is the pie chart</p>
       <div id="chart3" style="height:300px; width:650px;"></div>
    </div>
</div>
```

Figure 16-1 shows the final result.

Figure 16-1. *A page with three tabs containing different charts*

jqPlot Charts on Accordions

Another commonly used type of jQuery container is the *accordion*. This time you'll put the previous three charts into accordions. The list of plug-ins to include with the web page remains the same as in the previous example. You need to make some changes in the CSS styles; there are specific CSS classes for accordions, and their attributes need to be specified. They are shown in Listing 16-7.

Listing 16-7. ch16_02.html

```
<style type="text/css">
.ui-accordion {
    width: 690px;
    margin: 2em auto;
}
.ui-accordion-header {
    font-size: 12px;
}
.ui-accordion-content {
    font-size: 14px;
}
.jqplot-target {
    font-size: 18px;
}
ol.description {
    list-style-position: inside;
    font-size: 15px;
    margin: 1.5em auto;
    padding: 0 15px;
    width: 600px;
}
.section {
      width: 400px;
      height: 200px;
      margin-top: 20px;
      margin-left: 20px;
}
</style>
```

As you did in the previous example, you must create the jQueryUi widget. You can do this by calling the `accordion()` function:

```
$("#accordion").accordion();
```

You also need to bind this accordion to your charts, as shown in Listing 16-8. When you select an accordion tab, the event makes sure that the respective chart is redrawn inside it, calling the `replot()` function.

Listing 16-8. ch16_02.html

```
$('#accordion').bind('accordionchange', function(event, ui) {
    var index = $(this).find("h3").index ( ui.newHeader[0] );
    if (index === 0) {
       plot1.replot();
    }
```

```
    else if (index === 1) {
       plot2.replot();
    }
    else if (index === 2) {
       plot3.replot();
    }
});
```

As you can see, the way in which you define the accordions is very similar to the way you define the tabs. In the same way, you now define the elements that will be converted into accordion tabs in the HTML code (see Listing 16-9).

Listing 16-9. ch16_02.html

```
<div id="accordion" style="margin-top:50px">
  <h3><a href="#">Section 1</a></h3>
  <div>
    <p>This is the bar chart</p>
    <div class="section" id="chart1" data-height="200" data-width="400"></div>
  </div>

  <h3><a href="#">Section 2</a></h3>
  <div>
    <p>This is the multiseries line chart</p>
    <div class="section" id="chart2" data-height="200" data-width="400"></div>
  </div>

  <h3><a href="#">Section 3</a></h3>
  <div>
    <p>This is the pie chart</p>
    <div class="section" id="chart3" data-height="200" data-width="400"></div> </div>
</div>
```

As you can see in Figure 16-2, the result is similar to the previous one, but this time the different charts are replaced by sliding the accordion tab vertically.

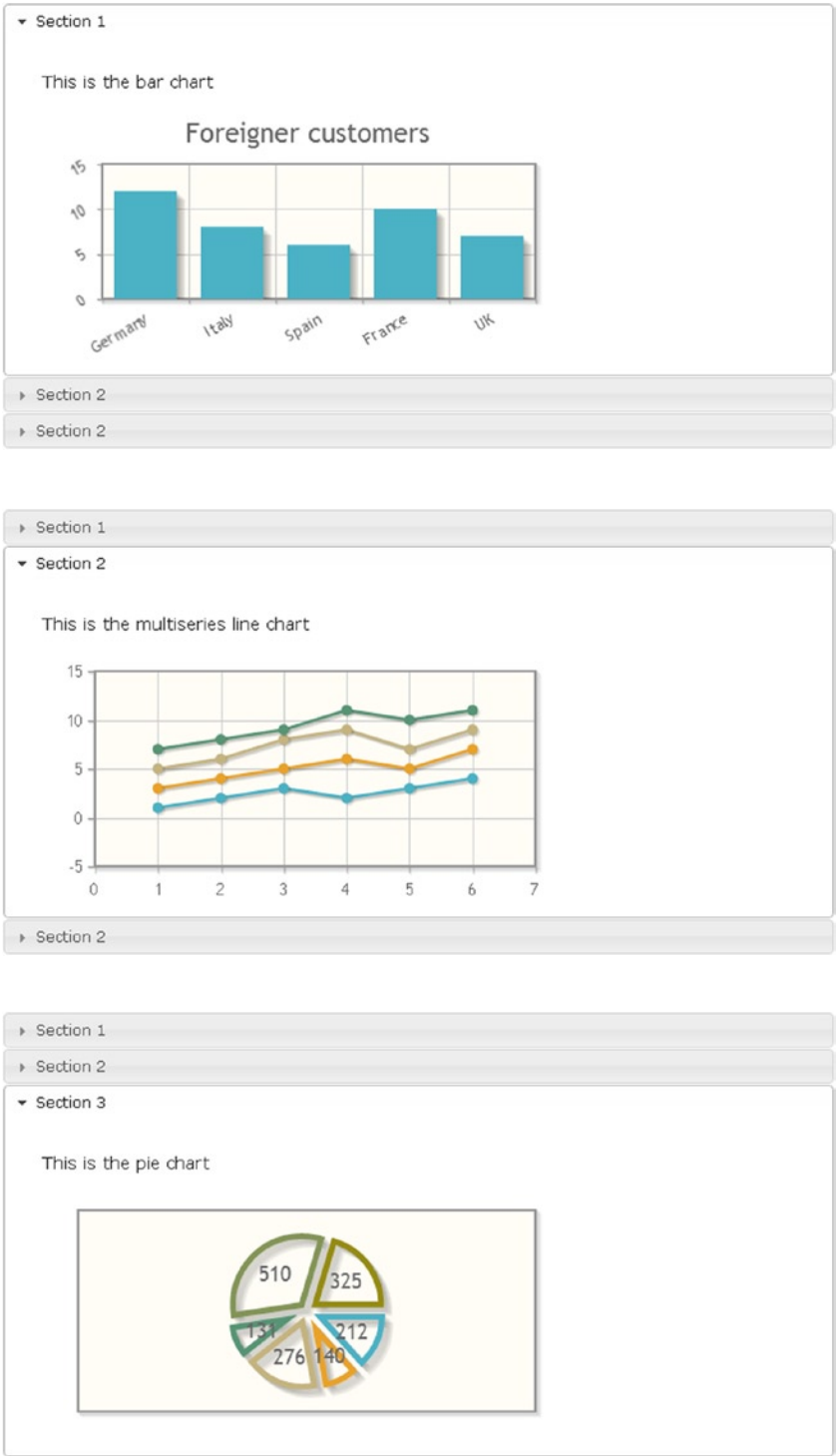


Figure 16-2. *An accordion widget containing three charts*

Resizable and Draggable Charts

Two other features that you can widely exploit in your charts enable users to resize and drag the container area. A resizable frame within a web page allows you to arbitrarily change its size and the size of the objects it contains. This feature could be combined with the ability to drag elements within the page, which would enable them to occupy different positions relative to the original.

In addition to giving fluidity to the layout of the page, this feature can sometimes be useful when you want the user to interactively manage spaces occupied by different frames on the page (see Figure 16-3).

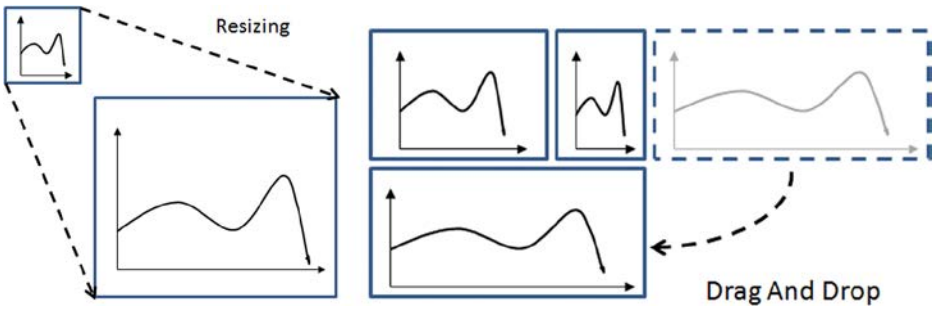


Figure 16-3. *Enclosing the charts in jQueryUI containers enables you to resize and move them around the page*

In this section, you'll see two examples. In the first example, you will focus on the resizing applied to a line chart. You'll see how easy it is to resize a chart contained within a container. In the second example, you'll further develop the example by adding two more line charts. Once the draggable property is enabled for all three charts, you will see how you can change their positions to your liking, or even exchange them.

A Resizable Line Chart

In this example you'll use a simple line chart. Thus, you'll no longer need to include all of the jqPlot plug-ins, except those needed for the jQuery container and the basic jqPlots library:

```
<!--[if lt IE 9]>
<script type="text/javascript" src="../src/excanvas.js"></script>
<![endif]-->
<script type="text/javascript" src="../src/jquery.min.js"></script>
<script type="text/javascript" src="../src/jquery.jqplot.min.js"></script>
<link rel="stylesheet" type="text/css" href="../src/jquery.jqplot.min.css" />
<link rel="stylesheet" href="../src/css/smoothness/jquery-ui-1.10.3.custom.min.css" />
<script src="../src/js/jquery-ui-1.10.3.custom.min.js"></script>
```

Or if you prefer to use a content delivery network (CDN) service:

```
<!--[if lt IE 9]><script type="text/javascript"
src="http://cdn.jsdelivr.net/excanvas/r3/excanvas.js"></script><![endif]-->
<script src="http://code.jquery.com/jquery-1.9.1.min.js"></script>
<script type="text/javascript" src="http://cdn.jsdelivr.net/jqplot/1.0.8/jquery.jqplot.min.js"></script>
<link rel="stylesheet" type="text/css" href="http://cdn.jsdelivr.net/jqplot/1.0.8/jquery.jqplot.min.css" />
<link rel="stylesheet" href="http://code.jquery.com/ui/1.10.3/themes/smoothness/jquery-ui.css" />
<script src="http://code.jquery.com/ui/1.10.3/jquery-ui.min.js"></script>
```

Even here it is necessary to specify some CSS styles, as shown in Listing 16-10.

Listing 16-10. ch16_03.html

```
<style type="text/css">
.chart-container {
    border: 1px solid darkblue;
    padding: 30px 0px 30px 30px;
    width: 900px;
    height: 400px;
}
#chart1 {
    width: 96%;
    height: 96%;
}
</style>
```

To the `<body>` part of the web page, you now add the `<div>` element, which will be the container enclosing the line chart called `chart1` (see Listing 16-11).

Listing 16-11. ch16_03.html

```
<div class="chart-container">
    <div id="chart1"></div>
</div>
```

Now, after you have defined the `chart-container` as container, you can handle it with two jQuery methods—the `resizable()` function adds the resizable functionality and the `bind()` function binds the event of resizing to the replotting of the chart (see Listing 16-12).

Listing 16-12. ch16_03.html

```
$(document).ready(function(){
    var plot1 = $.jqplot ('chart1', [[100, 110, 140, 130, 80, 75, 120, 130, 100]]);

    $('div.chart-container').resizable({delay: 20});
    $('div.chart-container').bind('resize', function(event, ui) {
       plot1.replot();
    });
});
```

The result is a resizable chart, shown in Figure 16-4, with a small grey triangle in the bottom-right corner. By clicking on it, the user can resize the container and consequently the jqPlot chart.

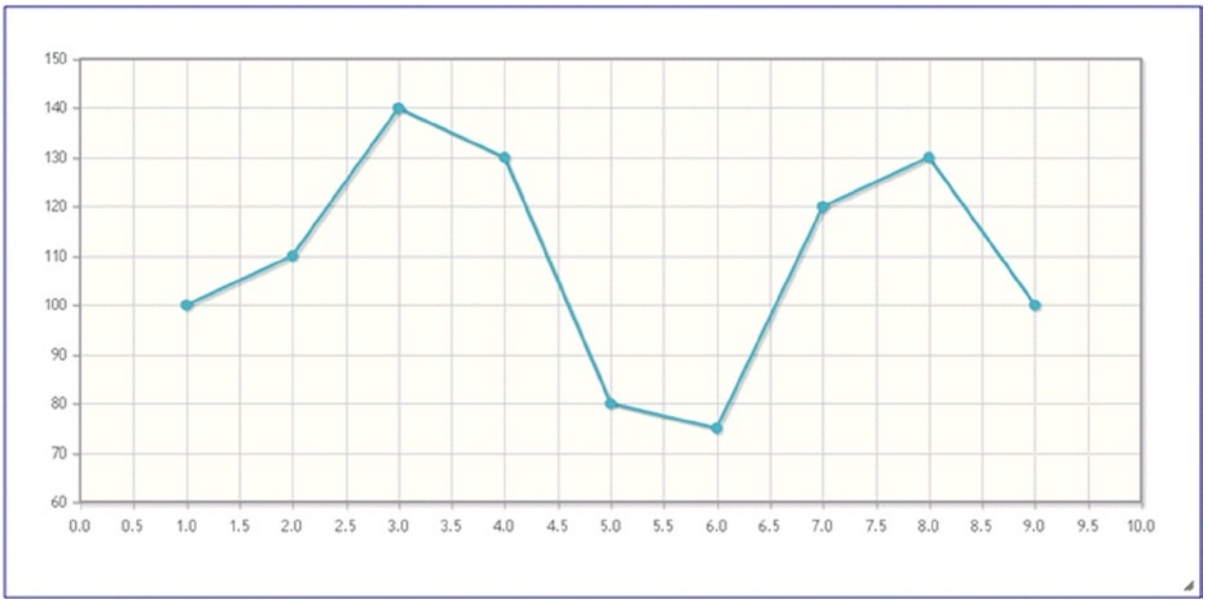


Figure 16-4. *A resizable line chart*

Three Draggable Line Charts

Starting from the previous example, you will add two more line charts by placing them in two independent containers. The goal here—in addition to making all three containers resizable—is to make the containers draggable. The final result is a web page with three line charts, the position of which can be changed by dragging them, even exchanging their positions.

Start by making some small additions to the previous example. In Listing 16-13, you add the other two containers (`chart-container2` and `chart-container3`) with the new line charts inside, naming them `chart2` and `chart3`, respectively.

Listing 16-13. ch16_03b.html

```
<BODY>
<div class="chart-container">
    <div id="chart1"></div>
</div>
<div class="chart-container2">
    <div id="chart2"></div>
</div>
<div class="chart-container3">
    <div id="chart3"></div>
</div>
</BODY>
```

Now that you have created the container for the new line chart, it is necessary to define them through three distinct `$.jqplot()` functions (see Listing 16-14). The values returned by these three functions will be passed to the three variables: `plot1`, `plot2`, and `plot3`. This is because you need to redraw each of the three charts whenever a change is made to the container, by using the `replot()` function on these three variables.

Listing 16-14. ch16_03b.html

```
$(document).ready(function(){
     var plot1 = $.jqplot ('chart1', [[100, 110, 140, 130, 80, 75, 120, 130, 100]],
        {seriesColors: [ "#bb0000" ]});
     var plot2 = $.jqplot ('chart2', [[120, 90, 150, 120, 110, 75, 90, 120, 110]],
        {seriesColors: [ "#00bb00" ]});
     var plot3 = $.jqplot ('chart3', [[ 130, 110, 140, 100, 80, 135, 120, 90, 110]],
        {seriesColors: [ "#0000bb" ]});

    $('div.chart-container').resizable({delay:20});
    ...
});
```

Now you'll activate the draggable feature for the three containers. Doing this is really quite simple; you need to add the function to the three jQuery selections applied to each container, as shown in Listing 16-15. Moreover, you'll add the resizing feature for the two new containers the same way as was done for the first container.

Listing 16-15. ch16_03b.html

```
$(document).ready(function(){
    ...
    var plot3 = $.jqplot ('chart3', [[130, 110, 140, 100, 80, 135, 120, 90, 110]],
       {seriesColors: ["#0000bb"]});

    $('div.chart-container').draggable({cursor: 'move'});
    $('div.chart-container2').draggable({cursor: 'move'});
    $('div.chart-container3').draggable({cursor: 'move'});

    $('div.chart-container').resizable({delay: 20});
    $('div.chart-container').bind('resize', function(event, ui) {
       plot1.replot();
    });
    $('div.chart-container2').resizable({delay: 20});
    $('div.chart-container2').bind('resize', function(event, ui) {
       plot2.replot();
    });
    $('div.chart-container3').resizable({delay: 20});
    $('div.chart-container3').bind('resize', function(event, ui) {
       plot3.replot();
    });
});
```

Nothing remains but to add CSS styles, thus defining the initial position and size of each container, as shown in Listing 16-16.

Listing 16-16. ch16_03b.html

```
<style type="text/css">
.chart-container {
    border: 1px solid darkblue;
    padding: 30px 0px 30px 30px;
    width: 300px;
```

```
    height: 200px;
    position: relative;
    float: left;
}
.chart-container2 {
    border: 1px solid darkblue;
    padding: 30px 0px 30px 30px;
    width: 200px;
    height: 200px;
    position: relative;
    float: left;
    margin-left: 20px;
}
.chart-container3 {
    border: 1px solid darkblue;
    padding: 30px 0px 30px 30px;
    width: 500px;
    height: 200px;
    position: relative;
    float: left;
    margin-left: 20px;
}
#chart1 {
    width: 96%;
    height: 96%;
}
#chart2 {
    width: 96%;
    height: 96%;
}
#chart3 {
    width: 96%;
    height: 96%;
}
</style>
```

In Figure 16-5, you can see the page layout when the page is initially loaded. Figure 16-6 shows a situation in which the user has changed the position and the size of the third chart to align it below the other two.

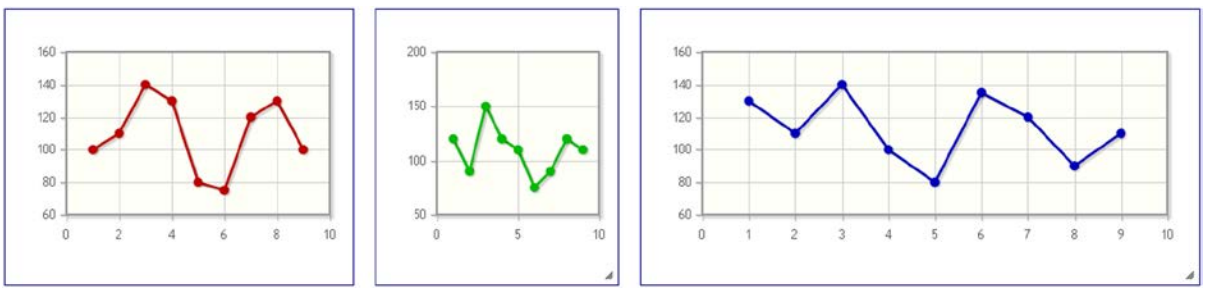


Figure 16-5. *The web page shows the three line charts enclosed in three different containers*

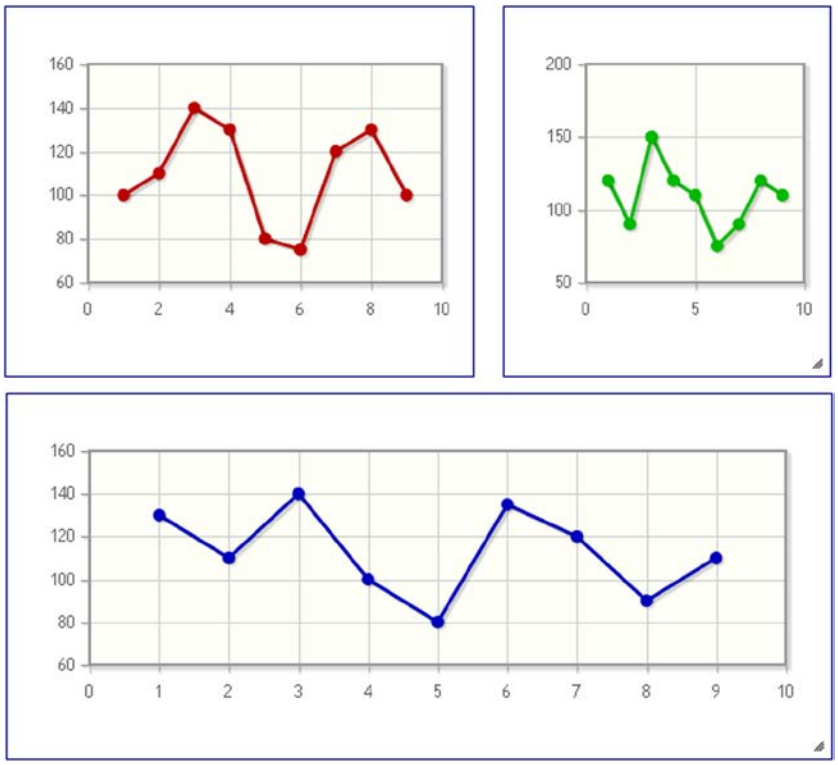


Figure 16-6. *By dragging and resizing the containers, the original layout can be changed*

Summary

In this chapter you have seen how to exploit the potential of widgets that the jQuery UI library makes available to you, widgets that help you improve the way your charts are represented. You have seen how to enclose more charts inside **containers, such as accordions and tabs**, so that you can view them one by one, even when they occupy the same area. You have also seen how to **resize** these **containers,** extending such capability to the charts developed with the jqPlot library.

So far you have deepened the graphical and representational aspects of your chart. In the next chapter, you'll learn about the core of your charts: **data management**. So far the variety of data defined in the page has been limited, in order to make the examples easier to comprehend. In reality, it is very unlikely that data will be defined on the same web page that contains the code management chart. More likely, the data are provided by external files or by databases through SQL queries.

CHAPTER 17

Handling Input Data

Once you have dealt with all the graphical aspects of a chart, it is time to analyze input data in more detail. In the previous chapters, you assigned the values of input data to arrays. These arrays were defined in the same HTML page within which the jqPlot code resides. You have frequently used these two ways:

```
var plot1 = $.jqplot ('chart1', [[100, 110, 140, 130, 80, 75, 120, 130, 100]]);
```

and

```
var data = [[100, 110, 140, 130, 80, 75, 120, 130, 100]];
```

In actuality, it is often necessary to interface with other technologies in order to obtain such data, and to do so you need to find a way that is well suited to any source of data. The need to use a common text format that can be easily handled by different scripting languages (especially JavaScript) and that remains comprehensible to humans, led to the use of the JavaScript Object Notation (JSON) format. You have briefly read about this kind of format in Chapter 1, but now you'll see how to use it concretely to handle input data from external sources.

This chapter studies in detail the JSON format, first illustrating the structure of the data in this format and then showing you how to use them with the jqPlot library. To this purpose, you'll see two different ways to handle JSON data—the first makes use of a jqPlot plug-in and the second uses a jQuery function that specializes in parsing JSON data.

Regardless of how the data coming from an external source are structured, if you want to have a complete overview of the management and handling of real data, you also need to take into account how this data are generated and the consequent acquisition mode. Therefore, in the last part of the chapter, you'll develop a real-time chart exclusively using the jqPlot library.

In fact, regardless of the format of the input data, many times the data source is not only external, but it is also continuous—the input data consists of a train of data in which the values are produced one at a time, serially and uninterrupted. Hence, the chart that will display this type of data will not only have to manage a format of data coming from an external source but needs also to be able to update itself continuously, thereby ensuring that the data representation (in this case, the real-time chart) is always updated.

Using the JSON Format

This section covers the JSON format, including various options for using it with the library jqPlot. First of all, you will learn how there can be structured data in the JSON format, by analyzing some syntax diagrams. Then you will move on to practical examples.

The JSON Format

JSON is a data exchange format. Thanks to its tree structure, in which each element is referred as a name-value pair, it is easy for humans to read and write it and for machines to parse and generate it. This is the main reason for its increasingly prevalent use.

The JSON structure is built on the combination of two different structures: arrays and objects (see Figure 17-1). Within them you can define all of the classic primitive values commonly used, even in other languages: numbers, Booleans, strings, and null value. This allows values contained in it to be exchanged between various programming languages. (At `www.json.org`, you can find a list of all languages that handle the JSON format, along with a list of all the related technologies, such as libraries, modules, plug-ins, and so on.)

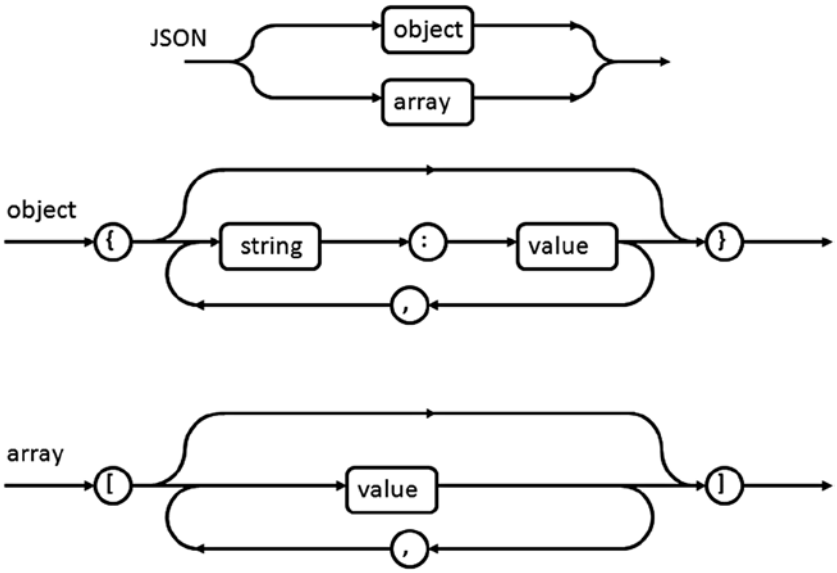


Figure 17-1. *Syntax diagrams for JSON*

Just to understand the syntax diagrams in Figure 17-1 better, you can analyze how a JSON format is structured. You must take into account two things. The first is that both the objects and the arrays contain a series of values identified by the `value` labels in the diagrams. `value` refers to any type of value, such as a string, a number, or a Boolean, and it can even be an object or an array.

In addition to this, you can easily guess that the JSON structure is a tree structure with different levels. The tree will have as nodes either arrays or objects; the leaves are the values contained in them.

Consider some examples. If you have a JSON structure with only one level, you will have only two possibilities:

- An array of values
- An object with values

If you extend the structure to two levels, the possibilities are four (assuming for simplicity that the tree is symmetrical):

- An array of arrays
- An array of objects
- An object with arrays
- An object with objects

And so on; the cases gradually become more complex.

The classic JSON structure is precisely the structure of the jqPlot library `options` object that you have already used frequently in this book. In fact, you have seen that, thanks to the objects that have a string associated with each value, these tree structures can describe any type of element. Even very complex elements, such as charts, can be easily understood and manipulated.

A Practical Case: The jqPlot Data Renderer

Referring to JSON as an exchange format, this section considers the simple case in which the external data source is a text file. In this example, you will use data rendered directly by the jqPlot library: the `json2` plug-in. This plug-in allows you to read the data in JSON format contained in a file in order to use them as input data. For your part, the only thing you are required to do is assign the external source to the `dataRenderer` property.

Start by implementing a blank HTML page, as shown in Listing 17-1.

Listing 17-1. ch17_01a.html

```
<HTML>
<HEAD>
    <TITLE>Chapter 17</TITLE>
<!--[if lt IE 9]>
<script type="text/javascript" src="../src/excanvas.js"></script>
<![endif]-->
<script type="text/javascript" src="../src/jquery.min.js"></script>
<script type="text/javascript" src="../src/jquery.jqplot.min.js"></script>
<link rel="stylesheet" type="text/css" href="../src/jquery.jqplot.min.css" />
<script>
$(document).ready(function(){
    //Add the JavaScript code here
});
</script>
</HEAD>
<BODY>
<div id="myChart" style="height:300px; width:500px;"></div>
</BODY>
</HTML>
```

In order to have the data renderer interpret the data, the external source must return a valid jqPlot data array. To extend the chart with this functionality, you need to include the `jqplot.json2` plug-in:

```
<script type="text/javascript" src="../src/plugins/jqplot.json2.min.js"></script>
```

Or use the content delivery network (CDN) service:

```
<script type="text/javascript"
    src="http://cdn.jsdelivr.net/jqplot/1.0.8/plugins/jqplot.json2.min.js"> </script>
```

As an external source, you can choose a TXT file containing the data. For this example, you'll create a new TXT file with Notepad or any other text editor. After you have edited the data as reported in Listing 17-2, save the file as `jsondata.txt`.

Listing 17-2. jsondata.txt

```
[[30, 12, 24, 54, 22, 11, 64, 33, 22]]
```

On the web page where you want to manage the data in an external file, you need to add the code in Listing 17-3.

Listing 17-3. ch17_01a.html

```
$(document).ready(function(){
    var ajaxDataRenderer = function(url, plot, options) {
       var ret = null;
       $.ajax({
          async: false,
          url: url,
          dataType:"json",
          success: function(data) {
             ret = data;
          }
       });
       return ret;
    };
    var jsonurl = "./jsondata.txt";
    var options = {
       title: "AJAX JSON Data Renderer",
       dataRenderer: ajaxDataRenderer,
       dataRendererOptions: {
          unusedOptionalUrl: jsonurl
       }
    };
    $.jqplot('myChart', jsonurl,options);
});
```

You'll get the chart shown in Figure 17-2, which derives its data directly from the TXT file.

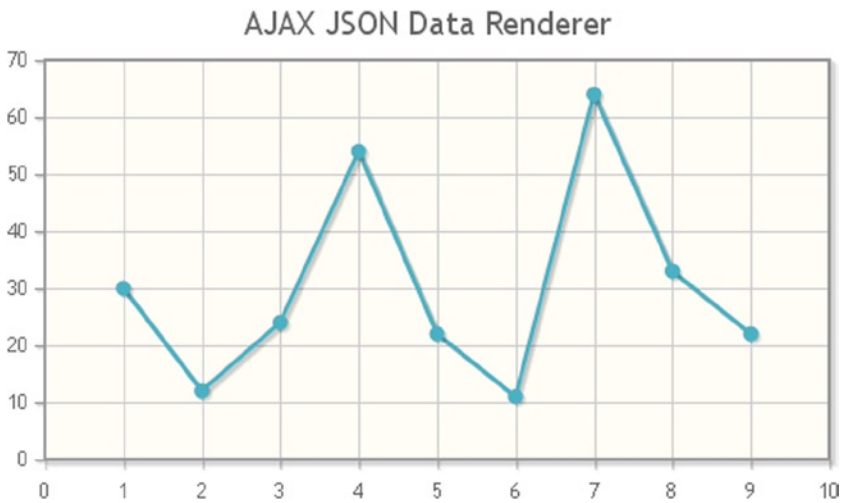


Figure 17-2. *A line chart representing data from the* `jsondata.txt` *file*

If you want to insert more than one series, the format of data within the TXT file remains the same as the format used for the input data arrays (see Listing 17-4). After you have copied this data in a file with an editor, save this file as `jsondata2.txt`.

Listing 17-4. jsondata2.txt

```
[[1,3,2,4,3,4,1,2],
[6,7,9,6,8,9,10,9],
[15,12,11,9,11,12,13,14]]
```

Figure 17-3 shows a chart with the three series read from the TXT file.

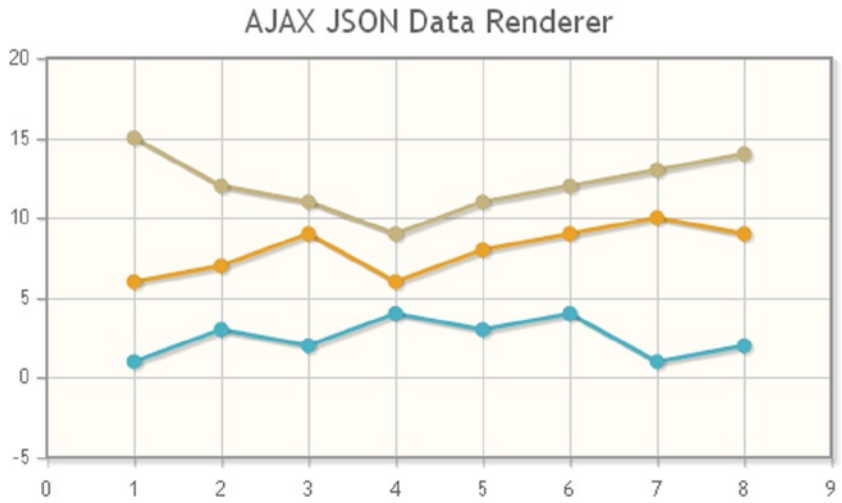


Figure 17-3. *A multiseries line chart representing data from a TXT file*

JSON and $.getJSON()

There is another way to use external JSON data in your jqPlot charts. Instead of using the `json2 jqPlot` plug-in as a data renderer, jQuery provides a method that performs the same functions; it's called `$.getJSON()`.

This method reads a JSON file and parses it. It also can load JSON-encoded data directly from a server by making an HTTP `GET` request. It is widely used in many applications on the Web, not only for jqPlot. This method has three arguments:

```
$.getJSON(url, data, success(data, textStatus, jqXHR));
```

Only `url` is mandatory; the other two arguments are optional. `url` is a string containing the URL of the JSON file or the URL of the server for the request. `data` is a string to be sent to the server with the request, and `success()` is a callback function that will be executed if the request succeeds.

The data contained in the file must follow the rules for JSON encoding. Because you are using them in order to be encoded by jqPlot, they should have this format:

```
{ "series_name1": [ value1, value2, value3, ...],
  "series_name2": [ value1, value2, value3, ...], ... }
```

Create a new TXT file and save it as `jsondata3.txt`. This file contains data from four distinct series, as shown in Listing 17-5.

Listing 17-5. jsondata3.txt

```
{"data1": [1,2,3,2,3,4],
 "data2": [3,4,5,6,5,7],
 "data3": [5,6,8,9,7,9],
 "data4": [7,8,9,11,10,11]}
```

The next step consists of the call to the `getJSON()` method:

```
$.getJSON('./jsondata3.txt', '', myPlot);
```

You must pay attention to write the right URL. In this example, the TXT file is in the same directory of the HTML file, so you need to add `./` as a prefix to the name of the file. The second argument is an empty string, because you do not need to send any data to the URL (it is a file, not a server application). `myPlot` is the returned value of the function, which checks if the loading of `$.jqplot()` is good. As you can see in Listing 17-6, you only need to add your own function, within which you have defined the `$.jqplot()` function. In this case, the data to pass—such as `data1`, `data2`, and so on—belongs to the data object and must be therefore passed as `data.data1`, `data.data2`, and so on.

Listing 17-6. ch17_02.html

```
$(document).ready(function(){
    var myPlot = function (data, textStatus, jqXHR) {
       $.jqplot ('myChart',[data.data1, data.data2, data.data3, data.data4]);
    };
    $.getJSON('./jsondata3.txt', '', myPlot);
});
```

You thus obtain a multiseries line chart (see Figure 17-4) as if you had written the data directly on the web page, but the chance to work with servers and other applications extends your capabilities enormously.

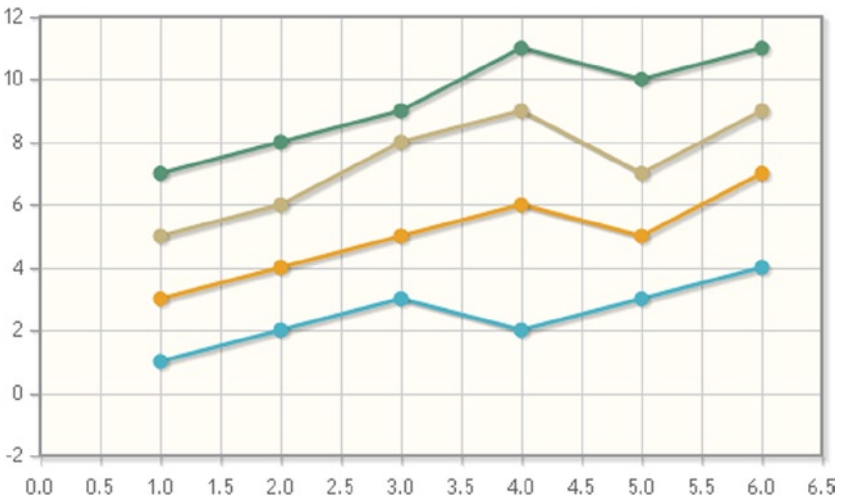


Figure 17-4. *A multiseries line chart representing data on a TXT file*

Real-Time Charts

Real-time charts automatically update themselves, thus allowing you to represent streams of data from a source that produces data continuously. This source can be a server, an application, a device connected to a PC, and so on. It is in such cases that a chart assumes the role of a true indicator, that is, a device which provides a visual indication of how a certain property varies over time.

You are now going to develop a simple real-time line chart using only the jqPlot library. Consider, for instance, that you want to implement an indicator of a magnitude that varies from 0 to 100%. This quantity could be, for example, the consumption of a resource (such as CPU), but can be applied to many other things such as temperature, the number of participants or connections, and so on. In this case, you'll start with the value of 50% and generate random variations in real time, simply for the sake of simulating a data source. It is possible to adapt this example to any other case, just by replacing the random function with a function that acquires data externally.

You'll implement a web page in which a line chart is represented, a chart in which there is only a small stretch set to the value of 50% (see Figure 17-5). This will be the starting point for the values of the streaming data.

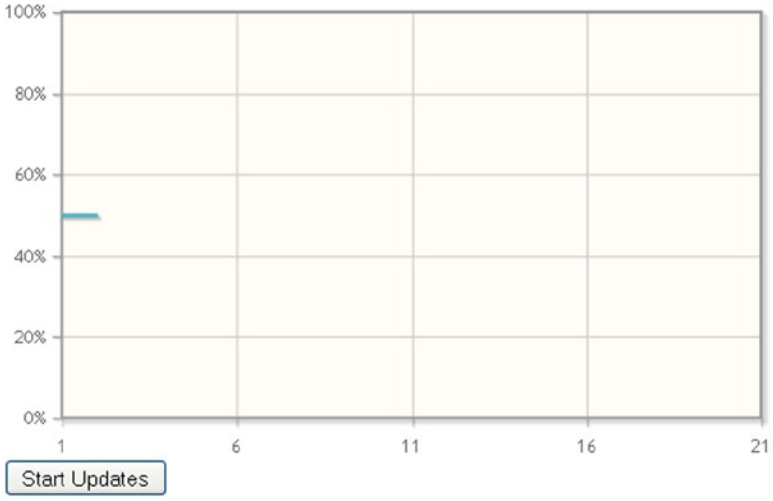


Figure 17-5. *The real-time chart before acquiring data shows only a small stretch as its starting point*

Listing 17-7 shows all that is needed to obtain the chart shown in Figure 17-5. In `options`, you define the limits of the range of the axes so that there are no variations during the chart update. To give the animation a more fluid effect, you eliminate the markers on the line and enable the `smooth` mode. Under the chart, you'll insert a button to start the update in real time.

Listing 17-7. ch17_03.html

```
<HTML>
<HEAD>
<TITLE>Real-time chart</TITLE>
<!--[if lt IE 9]>
<script type="text/javascript" src="../src/excanvas.js"></script>
<![endif]-->
<script type="text/javascript" src="../src/jquery.min.js"></script>
<script type="text/javascript" src="../src/jquery.jqplot.min.js"></script>
<link rel="stylesheet" type="text/css" href="../src/jquery.jqplot.min.css" />
```

```
<script class="code" type="text/javascript">
$(document).ready(function(){
    data = [50, 50];
    var options = {
       axes: {
          xaxis: {min: 1, max: 21, numberTicks: 5},
          yaxis: {min: 0, max: 100,numberTicks: 6,
             tickOptions:{formatString: '%d%'}
          }
       },
       seriesDefaults: {
          showMarker: false,
          rendererOptions: {smooth: true}
       }
    };
var plot1 = $.jqplot ('myChart', [data], options);
});
</script>
</HEAD>
<BODY>
<div id="myChart" style="height: 300px; width: 500px;"></div>
<button>Start Updates</button>
</BODY>
</HTML>
```

Inserting Listing 17-8, you now capture the `click` event of the button and link it to the execution of the `doUpdate()` function. Once the button is pressed, you can delete it from the web page.

Listing 17-8. ch17_03.html

```
$(document).ready(function(){
    ...
    var plot1 = $.jqplot ('myChart', [data],options);
    $('button').click( function(){
       doUpdate();
       $(this).hide();
    });
});
```

Hence, in Listing 17-9, you implement the function that generates random variations.

Listing 17-9. ch17_03.html

```
$(document).ready(function(){
    ...
    $('button').click( function(){
       doUpdate();
       $(this).hide();
    });
    function getRandomInt (min, max){
       return Math.floor(Math.random() * (max - min + 1)) + min;
    }
});
```

This function generates integer values between min and max values (which can be negative). These values are passed as arguments to the function. You set a possible variation between -3 and 3, which will be applied to the last values acquired. The real-time values are stored in an array called `data`, which operates as a sort of buffer. This array contains only 20 values, so that the first (the eldest) will be deleted and a new acquired value will be inserted in the last position of the array. As you can see in Figure 17-4, at the beginning you'll see an oscillating line that extends the length of the chart. Then the right end of the line will move, following the trend of the magnitude observed.

To obtain an animation you need to refresh the chart, so for each update you need to destroy the current chart (`plot1`), replace the data array with the new one, and then replot the whole `plot1` chart. At the end, you need to call the `setTimeout()` function, which will in turn call the `doUpdate()` function again. Thus, the cycle is repeated endlessly. You can update the chart every second (1,000 milliseconds), but these values, in other cases, will be chosen depending on the source data.

Go ahead and add Listing 17-10 to your code.

Listing 17-10. ch17_03.html

```
$(document).ready(function(){
    ...
    function getRandomInt (min, max) {
       return Math.floor(Math.random() * (max - min + 1)) + min;
    }
    function doUpdate() {
       var last = data[data.length-1];
       if(data.length > 19){
          data.shift();
       }
       var newlast = last + getRandomInt(-3, 3);
       if(newlast < 0)
          newlast = 0;
       data.push(newlast);
       if (plot1) {
          plot1.destroy();
       }
       plot1.series[0].data = data;
       plot1.replot( {resetAxes: true} );
       plot1 = $.jqplot ('myChart', [data], options);
       setTimeout(doUpdate, 1000)
    }
});
```

Figure 17-6 demonstrates how the real-time chart shows the stream of data varying around the value of 50%.

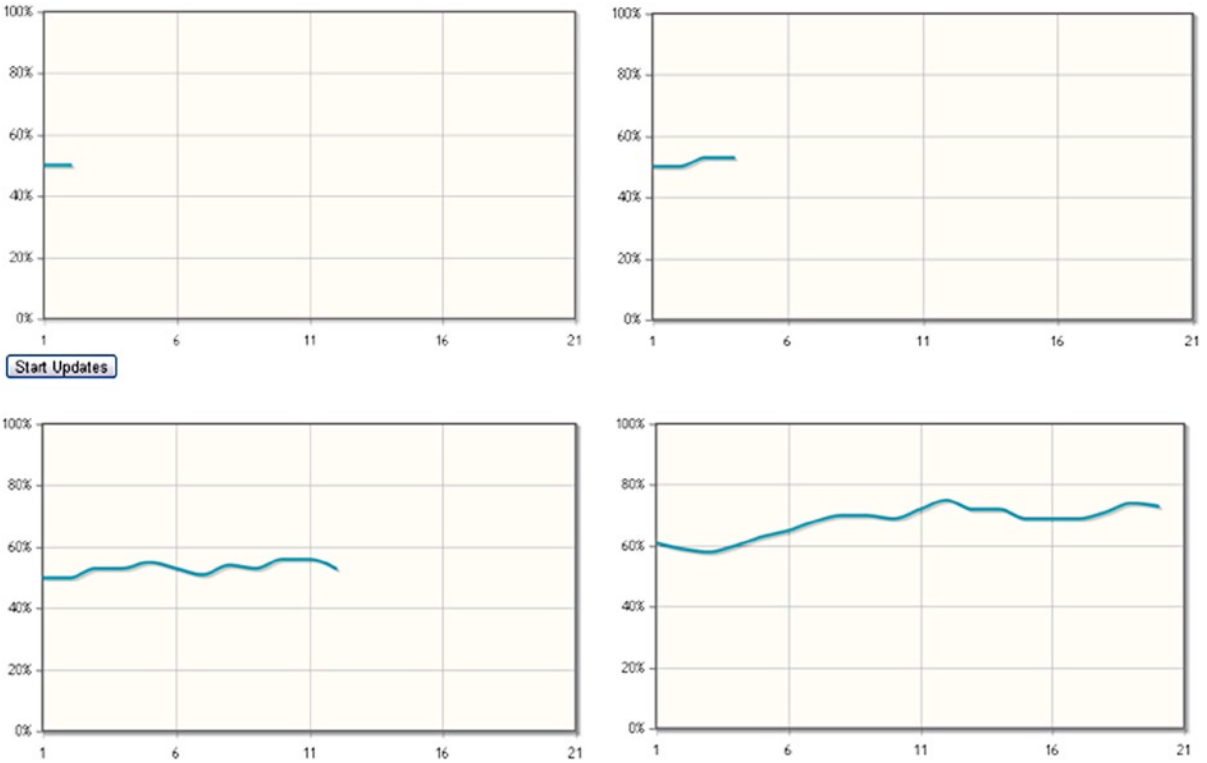


Figure 17-6. *A real-time line chart representing values moment by moment*

Summary

In this chapter you have seen how input data often come from external sources and how it is possible to handle them. In a particular way, you have seen how external data in **JSON format** can be used as input data arrays for the charts developed with the jqPlot library. In regard to the management of the data generated in real time, you have seen a simple but effective example in which you implemented a **real-time line chart** that is updated as the acquired data vary.

In the next chapter, you'll make the acquaintance of another library. In many ways, it's very similar to jqPlot and it's called **Highcharts**. Through many examples, you'll see how this commercial library, keeps many of the basic features of jqPlot, but also greatly expands your possibilities and adds more features for your charts. As you will see soon, the Highcharts library provides more functionality than you have seen so far.

CHAPTER 18

Moving from jqPlot to Highcharts

Following in the footsteps of the jqPlot framework, a new JavaScript library is catching up, called **Highcharts**. This is a commercial product and was completed in late 2009 by the Norwegian company Highsoft Solutions AS. As this book is being written, this new library is at version 3.0.1 and is increasingly being offered in the market as a new solution for the professional representation of charts.

Since we have been referring in many ways to the syntax and structure of jqPlot, it seemed appropriate to add the Highcharts library to the end of this first part of the book, dedicating a separate chapter to it. This framework has inherited all the advantages of jqPlot: it is possible to implement a complete chart on a web page by adding only a few lines of code. Highcharts is also capable of interfacing with many other JavaScript frameworks, including jQuery. Just like jqPlot, its syntax is simple, essential, and intuitive, but without precluding the possibility of further extensions and customizations. And that's not all.

Highcharts is a professional product, so it goes far beyond what the jqPlot library can offer. In this chapter you'll see how, starting with the cases you have already seen in jqPlot, it is possible to add more features to your charts. In fact, in the first examples you'll see how to expand the already good graphical capabilities that you've achieved thanks to the jqPlot library. You'll do this with further enhancements to the chart elements, such as tooltips, marker points, and the grid. Moreover, you'll discover how this library contains a number of themes and learn how you can apply them to your chart in order to give it a look that is always different. Another example illustrates how the Highcharts library reads data directly from a file. Finally, in the last part of the chapter, you'll take a step forward with the creation of three types of very particular charts, as follows:

- **The Master and Slave chart**—A very useful solution when you need to visualize a huge amount of data but want to analyze only one detail at a time.
- **The Gantt chart**—A chart that's particularly suited to the scheduling of processes and projects.
- **Combined chart**—A very rich visualization created to combine different types of charts.

These three charts types are possible thanks to the Highcharts library. So get ready to familiarize with this beautiful library at the exact point where you left off with jqPlot, by placing and gradually inserting these new features.

The Highcharts Distribution

Regarding the use of Highcharts, this product has both commercial and free, noncommercial licenses. Clearly, the free use of the Highcharts library is restricted only for personal and nonprofit purposes.

After downloading the package of files that make up the distribution, you can see that it contains a large group of files. (See Appendix A for further information.)

The main file is `highcharts.js`, which is the real core of the framework and it is essential to include this file in your web pages if you want to draw your chart with Highcharts. The way to include it is always the same, and you can do this in two ways: via the local method or the content delivery network (CDN) service.

For the local method, use the following:

```
 <script type="text/javascript" src="../src/js/highcharts.js"></script>
```

You have two CDN service options. If you want to have the latest stable version, write:

```
<script src="http://code.highcharts.com/highcharts.js"></script>
```

Otherwise, if you want to include a specific version, use the following:

```
<script src="http://code.highcharts.com/3.0.5/highcharts.js"></script>
```

Also, along with `highcharts.js` you also need to include jQuery since, like jqPlot, it is built upon that library, and therefore requires your access to it. Even in this case you can choose the local solution:

```
<script type="text/javascript" src="../src/js/jquery-1.9.1.js"></script>
```

or the CDN solution:

```
<script type="text/javascript" src="http://code.jquery.com/jquery-1.9.1.min.js">
</script>
```

However, it is also possible to work with other libraries as a base instead of jQuery, by using some adapter files contained within the distribution. Thus you can replace jQuery with libraries such as *MooTools* or *Prototype*.

When you delve further inside the distribution, you'll see another file called `highcharts-more.js`. This file contains all the functionalities concerning some extensions and special cases of the library such as, for example, those required to develop gauges, ranges, and polar charts. Furthermore, there is a group of files that plays a similar role to the plug-ins in jqPlots, since each of these files adds a specific functionality to your chart. You can find them in the directory `/js/modules`. Table 18-1 describes the module files.

Table 18-1. *The modules in the Highcharts distribution (v.3.0.5)*

Module	Description
`annotations.js`	A utility to add annotations to the chart elements.
`canvas-tools.js`	An additional file for Android 2.x devices that do not support SVG. It contains `canvg`, the JavaScript parser for SVG.
`data.js`	A utility to ease parsing of input sources, such as CSV, HTML tables, or grid views, into basic configuration options for direct use in the Highcharts constructor.
`exporting.js`	It allows users to download the chart as a PDF, PNG, JPEG, or SVG vector image.
`funnel.js`	Required for the drawing of funnel charts.
`Heatmap.js`	Required for the drawing of heatmaps.
`map.js`	Required for working on geographical maps.

Finally, you can find a directory called `/js/themes` containing the themes, which are described in Table 18-2. They are covered in more detail later, in a separate section.

Table 18-2. *The themes in the Highcharts distribution (v.3.0.5)*

Themes	Description
`dark-blue.js`	It adds a dark blue background to the chart. The colors of the series are vivid.
`dark-green.js`	It adds a dark green background to the chart. The colors of the series are vivid.
`gray.js`	It adds a black background to the chart. The colors of the series are vivid.
`grid.js`	It intensifies the grid below the chart. The colors of the series are vivid.
`skies.js`	It adds a sky with some clouds as background. The colors of the series are dark.

Similarities and Differences

This section explores the similarities and differences between the jqPlot library, which you now know well, and the new Highcharts library. You will find many points in common, both in the way you define the options structure and in many other aspects such as the data handling.

However, to better understand the similarities and differences between these two libraries, you will study a multiseries line chart, and using the same data set, you will generate two representations (one for each library). Thus, analyzing the two cases, you'll compare both the structure of the code and the representation itself.

Start with the example of the basic multiseries line chart that you saw earlier with jqPlot (see Listing 18-1).

Listing 18-1. ch18_01x.html

```
<script type="text/javascript" src="../src/jquery.min.js"></script>
<script type="text/javascript" src="../src/jquery.jqplot.min.js"></script>
<link rel="stylesheet" type="text/css" href="../src/jquery.jqplot.min.css" />
<script type="text/javascript">
$(document).ready(function(){
    var series1 = [1, 2, 3, 2, 3, 4];
    var series2 = [3, 4, 5, 6, 5, 7];
    var series3 = [5, 6, 8, 9, 7, 9];
    var series4 = [7, 8, 9, 11, 10, 11];
    $.jqplot ('myChart', [series1, series2, series3, series4]);
});
</script>
```

You can build its practical equivalent using the Highcharts library instead, as shown in Listing 18-2.

Listing 18-2. ch18_01a.html

```
<script type="text/javascript" src="../src/js/jquery-1.9.1.js"></script>
<script type="text/javascript" src="../src/js/highcharts.js"></script>
<script>
var series1 = [1, 2, 3, 2, 3, 4];
var series2 = [3, 4, 5, 6, 5, 7];
var series3 = [5, 6, 8, 9, 7, 9];
var series4 = [7, 8, 9, 11, 10, 11];
$(function () {
    $('#myChart').highcharts({
```

```
        chart: {
           type: 'line'
        },
        series: [{ data: series1 },
           { data: series2 },
           { data: series3 },
           { data: series4 }]
    });
});
</script>
```

The similarity in the syntax is pretty obvious. The structure, passed as an argument to the `highcharts()` function, is very similar to the structure that you used to define as `options` in jqPlot. In Highcharts, this structure is defined as a **configuration object**. Nothing will prohibit you from defining it externally via an `options` variable in a manner very similar to what you do with jqPlot, as shown in Listing 18-3.

Listing 18-3. ch18_01a.html

```
var options = {
    chart: {
       type: 'line'
    },
    series: [{ data: series1 },
       { data: series2 },
       { data: series3 },
       { data: series4 }]
}
$(function () {
    $('#myChart').highcharts(options);
});
```

Loading the same data, but using the two different libraries, you get the charts shown in Figure 18-1.

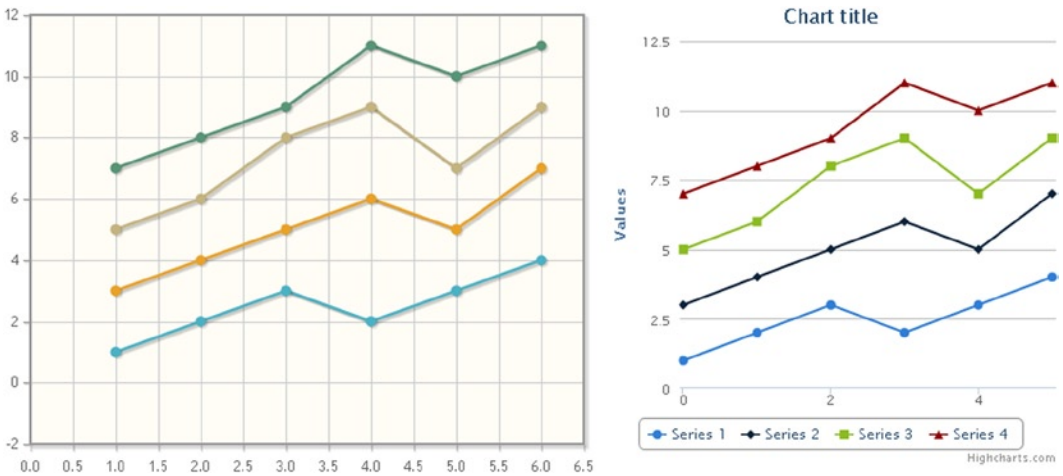


Figure 18-1. *Comparison between the two line charts generated with the jqPlot (left) and Highcharts (right) libraries*

From Figure 18-1, you can see that the layout of the two charts is quite different. Moreover, in its basic configuration, the Highcharts library provides a title, a legend, and axis labels, even though these were not specified explicitly. Furthermore, even tooltips are active, and in fact, if you move the mouse over the points of the lines, a tooltip will appear immediately, reporting information related to that point.

Figure 18-2. *By default, a tooltip reports the x and y values of the point and the name of the series to which it belongs*

In addition, while the page is loading the chart, it's drawn gradually from left to right with an animation. By default, a legend is also added, corresponding to the legend generated by the `enhancedLegendRenderer` plug-in of jqPlot (see the "Handling Legends" section in Chapter 10); in fact, the items inside the legend are active and if you click on them the corresponding series is deleted from the chart. In addition, a detail of no small importance, the scale of the axes and the grid are updated automatically. They adapt to the new range covered by the remaining sets in order to optimize the display. This is a gradual effect and is included by default.

So it is clear that the starting point of Highcharts is pretty much the culmination of jqPlot. In fact, as you'll see later, this library will further expand your capabilities, without the need to touch the code, or write additional functions or JavaScript libraries.

From the basic code, as in the case of `options` in jqPlot, the configuration object in Highcharts is made up of many components. Within these components there is a whole set of properties prepopulated with default values. If these properties are not explicitly specified, then the chart will follow the default values. The behavior is almost the same as in `options` for jqPlot. This provides a considerable advantage to the developer who makes use of these libraries. Although these libraries do not specify all attributes, they do represent the most common types of charts. This greatly facilitates a developer's work, especially when less experienced.

The component objects most commonly used within the object configuration are:

- **`chart`**—Where the properties are defined as the layout, events, animations, relative margins, and drawing area size
- **`series`**—Where you assign the arrays corresponding to the data series and their properties
- **xAxis and yAxis**—Where you define the properties for the axes and axis labels
- **title and subtitle**—Where you insert the title and the eventual subtitle of the chart
- **`legend`**—Where you specify everything about the legend
- **`tooltip`**—Where you specify everything about the tooltips

As you can see, all of these components are already familiar. You therefore won't examine them in more depth in this chapter. Those who wish to delve deeper into the components of the configuration object can refer to the full documentation on the official web site of the Highcharts library (`http://api.highcharts.com/highcharts`).

Line Charts with Highcharts

You started off by considering a simple line chart, to understand the similarities and differences between the jqPlot and Highcharts libraries. As was done for the jqPlot library, this section introduces the Highcharts library by treating the implementation of line charts in greater detail.

First, you'll complete the simple line chart by adding more elements. Later, you'll look at different ways in which you can define input data, especially by managing their categories and ranges. Hence, you'll look at further graphical aspects, such as the grid, tooltips, and the legend, and at how to customize these elements. Finally, you'll learn about the themes this library makes available, including what they are and how to use them.

Completing the Line Chart

The line chart you just created with Highcharts was generated without defining any property in the configuration object, it simply highlighted everything that comes by default from the Highcharts library. First, you'll define a set of properties that complete the chart, then you'll be adding more, to enrich the chart with some new features that are not always provided by the jqPlot library. So, as a first step, add the properties and their attributes that are highlighted in bold in Listing 18-4 to your basic code.

Listing 18-4. ch18_01b.html

```
<script type="text/javascript" src="../src/js/jquery-1.9.1.js"></script>
<script type="text/javascript" src="../src/js/highcharts.js"></script>
<script>
var series1 = [1, 2, 3, 2, 3, 4];
var series2 = [3, 4, 5, 6, 5, 7];
var series3 = [5, 6, 8, 9, 7, 9];
var series4 = [7, 8, 9, 11, 10, 11];
$(function () {
    $('#myChart').highcharts({
       chart: {
          type: 'line',
          marginTop: 60,
          marginLeft: 60
       },
       title: {
          text: 'Central Art Museum',
          x: 10
       },
       subtitle: {
          text: '27th April,2013',
          x: 10
       },
       xAxis: {
          title: {
          text: 'Time',
          x: 100
       },
       categories:
         ['9:00', '11:00', '13:00', '15:00', '17:00', '19:00'],
         tickmarkPlacement: 'on'
       },
       yAxis: {
          title: {
             text: 'Visitors'
          }
       },
```

```
        series: [{
           data: series1,
           name: 'Australia'
        }, {
           data: series2,
           name: 'Belgium'
        }, {
           data: series3,
           name: 'Canada'
        }, {
           data: series4,
           name: 'Danmark'
        }],
        legend: {
           layout: 'vertical',
           align: 'right',
           verticalAlign: 'top',
           y: 60
        }
    });
});
</script>

<div id="myChart" style="width: 600px; height: 400px;"></div>
```

You'll get the line chart in Figure 18-3.

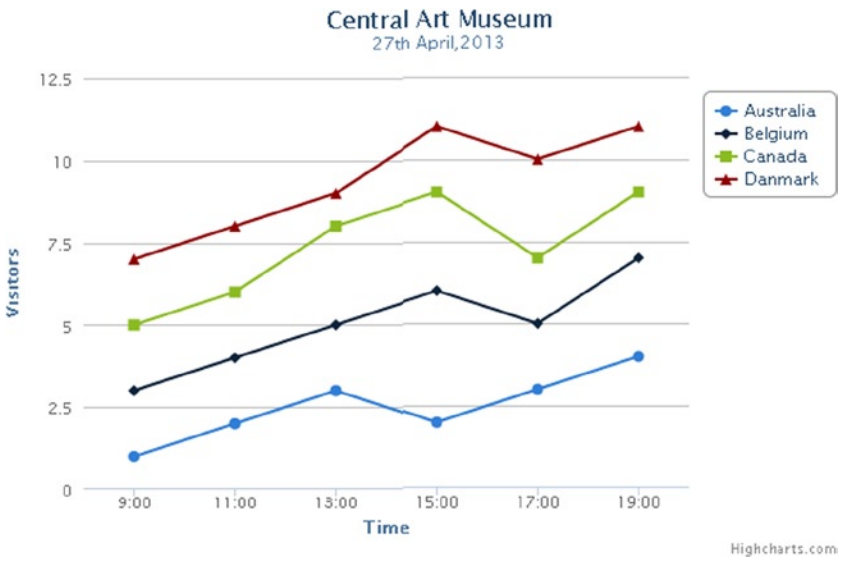


Figure 18-3. *A complete line chart with the Highcharts library*

Different Ways of Handling Input Data

Generally, input data with two coordinates (x, y) can be treated in two different ways. If the sequence of the values of x is the same for all series, as the example in question, it is preferable to handle the coordinates separately. You define an array of y values for each series, and an array containing all the values x. The y arrays are passed to as many **data** properties, within the `series` component, specifying their name with a string in the `name` property. Instead the x array is passed to the `categories` property in `xAxis`. In the second way, you directly define arrays containing both values, x and y. Since this choice is more laborious, it is preferable when the series do not share the same values on any of the two coordinates. However, by way of example, you would have had the same results if you had defined the points as in Listing 18-5.

Listing 18-5. ch18_01c.html

```
var serie1 = [ [Date.UTC(2013, 3, 27, 9), 1],
               [Date.UTC(2013, 3, 27, 11), 2],
               [Date.UTC(2013, 3, 27, 13), 3],
               [Date.UTC(2013, 3, 27, 15), 2],
               [Date.UTC(2013, 3, 27, 17), 3],
               [Date.UTC(2013, 3, 27, 19), 4] ]

var serie2 = [ [Date.UTC(2013, 3, 27, 9), 3],
               [Date.UTC(2013, 3, 27, 11), 4],
               [Date.UTC(2013, 3, 27, 13), 5],
               [Date.UTC(2013, 3, 27, 15), 6],
               [Date.UTC(2013, 3, 27, 17), 5],
               [Date.UTC(2013, 3, 27, 19), 7] ];

var serie3 = [ [Date.UTC(2013, 3, 27, 9), 5],
               [Date.UTC(2013, 3, 27, 11), 6],
               [Date.UTC(2013, 3, 27, 13), 8],
               [Date.UTC(2013, 3, 27, 15), 9],
               [Date.UTC(2013, 3, 27, 17), 7],
               [Date.UTC(2013, 3, 27, 19), 9] ];

var serie4 = [ [Date.UTC(2013, 3, 27, 9), 7],
               [Date.UTC(2013, 3, 27, 11), 8],
               [Date.UTC(2013, 3, 27, 13), 9],
               [Date.UTC(2013, 3, 27, 15), 11],
               [Date.UTC(2013, 3, 27, 17), 10],
               [Date.UTC(2013, 3, 27, 19), 11] ];
$(function () {
    $('#myChart').highcharts({

...

    xAxis: {
       title: {
          text: 'Time',
          x: 100
       },
```

```
        type: 'datetime',
        //categories: ['9:00', '11:00', '13:00', '15:00', '17:00', '19:00'],
        tickmarkPlacement: 'on'
    },
...
    });
});
```

This example is a good way to introduce a method that is used very frequently in Highcharts when you have to deal with `datetime` data: `Date.UTC()`. This is a JavaScript function that gives the number of milliseconds passed from 0:00 on January 1, 1970 to the date passed as an argument. The number of arguments that it can accept is variable, depending on how precise the date should be:

```
Date.UTC( years, months, days, hours, minutes, seconds)
```

If you want to specify only the days of a year, it is sufficient to write only the first three arguments:

```
Date.UTC(2012, 4, 27)
```

If, as in this example, you are talking about hours, then you need to specify four arguments:

```
Date.UTC(2013, 3, 27, 11)
```

■ **Note** The months' range is 0-11 (not 1-12), the days' range is 1-31 days (as usual), and the hours' range is 0-23. That is why April is indicated with the number 3 in the example.

You have just seen the two ways to pass input data in the Highcharts library. But what you'll be using in the following examples is a third option, which allows you to define a range of values on the x-axis without defining any array of categories (in my opinion, this choice is better suited to a bar chart than to a line chart, but it is important to deal with this now, in order to get the complete picture of the situation). Indeed, in a line chart the values on the x-axis increase based on a very precise scale. Highcharts provides a component called `plotOptions`, which allows you to define two particular properties: `pointInterval` and `pointStart`. With `pointInterval`, you can define the interval between one point and the next, and with `pointStart`, you define the starting value of the scale on x. Because you have to deal with the hours of a day on the x-axis, you have a data type `datetime`. Therefore, in order to obtain the same result on the x-axis without specifying the values passed to `categories`, you add the new component to the configuration object, as shown in Listing 18-6.

Listing 18-6. ch18_01d.html

```
plotOptions: {
    line: {
        pointInterval: 2 * 3600 * 1000,// h * m* s hour
        pointStart: Date.UTC(2013, 3, 27, 9, 0, 0)
    }
},
```

The grid: Advanced Management

One of the features that makes Highcharts a more advanced library than jqPlot is the management of the grid underlying the chart. In fact, this library provides you with some properties that allow you to alternate different types of grid layout on the same chart, creating pleasing and innovative effects. You'll see how this is possible by modifying the example you've been working on (see Listing 18-7).

Listing 18-7. ch18_01e.html

```
xAxis: {
    title: {
       text: 'Time',
       x: 100
    },
    type: 'datetime',
    gridLineWidth: 1,
    gridLineDashStyle: 'dot',
    minPadding: 0.1,
    maxPadding: 0.1,
    tickInterval: 4 * 3600 * 1000
},
yAxis: {
    title: {
       text: 'Visitors'
    },
    tickInterval: 4,
    gridLineColor: '#618661',
    minorTickInterval: 2,
    minorGridLineColor: '#618661',
    minorGridLineDashStyle: 'dashdot',
    alternateGridColor: {
       linearGradient: {
          x1: 0, y1: 1,
          x2: 1, y2: 1
       },
       stops : [
          [0, "#F8F8EE"],
          [1, "#A2B9A6"]
       ],
    },
    lineWidth: 1,
    lineColor: '#CACACA',
    tickWidth: 2,
    tickLength: 4,
    tickColor: '#CACACA'
},
```

In addition to no longer having straight lines connecting the dots, but rather curves (corresponding to what you obtained with `smooth` in jqPlot), you need to change the type of chart from `line` to `spline` (see Listing 18-8).

Listing 18-8. ch18_01e.html

```
chart: {
    type: 'spline',
    marginTop: 60,
    marginLeft: 60
},

plotOptions: {
    spline: {
       pointInterval: 2 * 3600 * 1000,// h * m* s hour
       pointStart: Date.UTC(2013, 3, 27, 9, 0, 0)
    }
},
```

From these changes you get the chart shown in Figure 18-4.

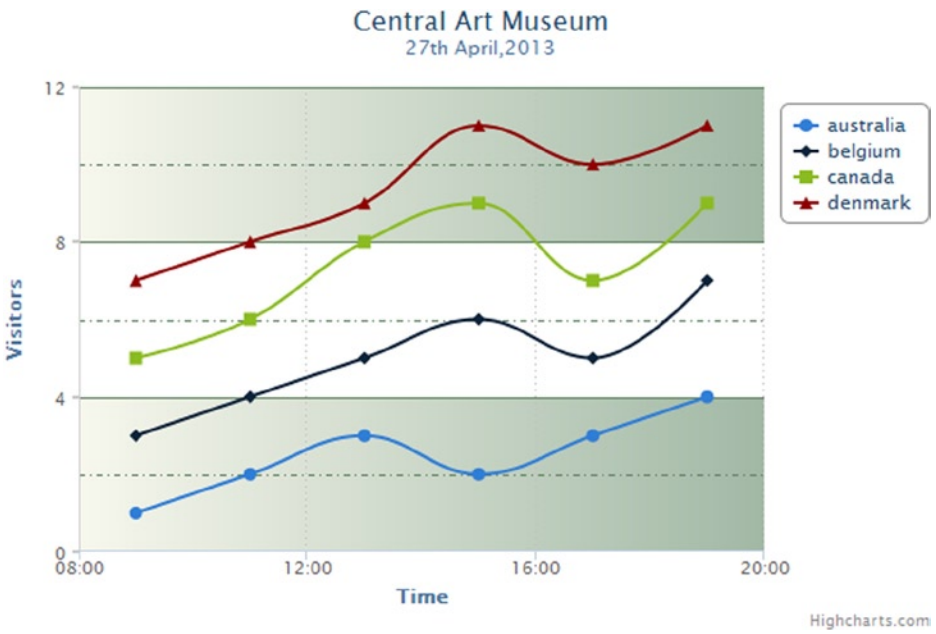


Figure 18-4. *A grid with alternating colors, a dash style, and a gradient effect*

There are two different classes of properties for grid lines, beginning with `grid-` and `-minorGrid,` and they can be defined both on the x-axis and on the y-axis, independently. This division into two different classes allows you to define two different types of grids for each axis, which will alternate between them line by line, giving a richer and more diversified chart.

Also with regard to the areas that are defined between the lines of the grid, it is possible to apply alternating colors, even with a gradient whose orientation can be adjusted. This is all thanks to the `alternateGridColor` property, which contains two additional properties to define. With `linearGradient` it is possible to give the gradient an orientation—0 and 1 specify the two extremes of the gradient on x and y. With `stops`, you can define the colors assigned to 0 and 1.

Customizing Tooltips with HTML

What in jqPlot constituted a real customization, here is formalized through the use of four properties defined within the `tooltip` component. `useHTML`, which if set to `true`, activates the construction mode of the new tooltip structure through HTML tags. With the `headerFormat`, `pointFormat`, and `footerFormat` properties, you can define the header, the body, and the tail of the tooltip, respectively. Dynamic values that change from point to point are accessible by including them in braces in the HTML structure. With `{ series.color }` and `{ series.name }`, you can access the color and the name of the series, whereas with `{ point.x }` and `{ point.y }`, you can access the x- and y-values of the point (see Listing 18-9).

Listing 18-9. ch18_01e.html

```
tooltip: {
    useHTML: true,
    headerFormat: '<small>{point.key}</small><table>',
    pointFormat: '<tr><td style="color: {series.color}">' +
       '<img src="flags/{series.name}.png" '+
       'height="22" width="32"> {series.name}:</td>' +
       '<td style="text-align: right"><b>{point.y}</b></td></tr>',
    footerFormat: '</table>'
},
```

■ **Note** The PNG files required to show the flags in the tooltips are included in the source code which is available on the Source Code/Downloads tab of the book's Apress product page (www.apress.com/9781430262893).

Figure 18-5 shows the result of what you have just done.

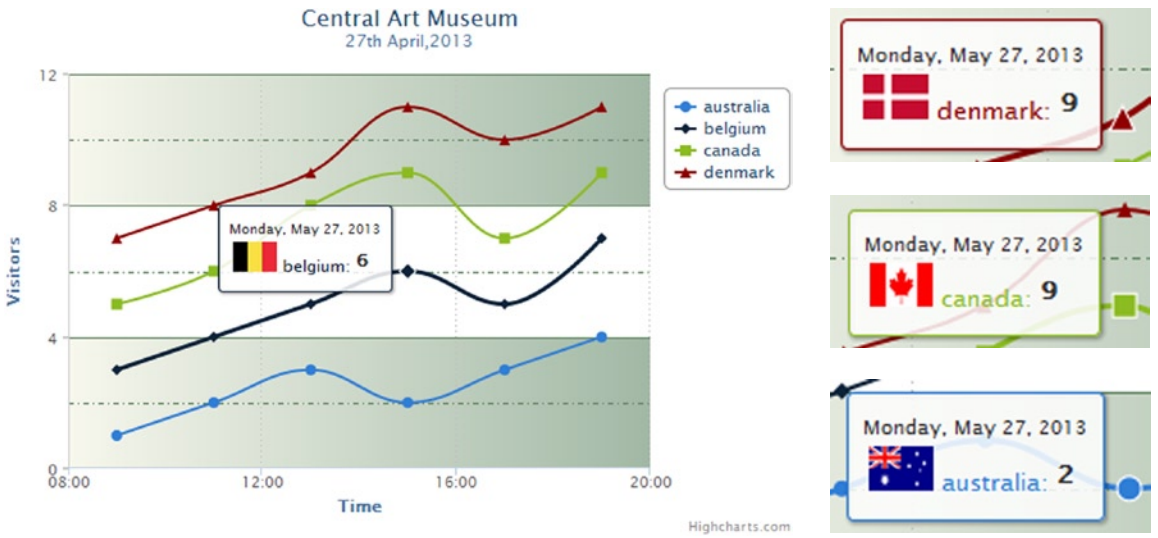


Figure 18-5. *With the HTML customization of the tooltips, it is possible to modify their default layout*

Customizing the Legend with HTML

The same type of customization is also applicable to the legend, as shown in Figure 18-6.

Figure 18-6. *A legend, customized using HTML with images and colors*

It is necessary to enable the `useHTML` property here also, and then to write a function that returns a string containing the HTML code to the `labelFormatter` property (see Listing 18-10).

Listing 18-10. ch18_01f.html

```
legend: {
    layout: 'vertical',
    align: 'right',
    verticalAlign: 'top',
    y: 60,
    useHTML: true,
    labelFormatter: function () {
       return '<table><tr><td style="color: '+this.color +
       '"><img src="flags/'+ this.name +
       '.png" height="15" width="20"><b> ' +
       this.name;+'</b></td></tr></table>'
    }
},
```

Adding Bands

Colored bands can be used to highlight or define regions in your chart. This functionality is fully integrated in Highcharts in the `plotBands` component and can be implemented with a very few lines. This component takes an array of bands, and you can define all of its properties within each band. Depending on where `plotBands` is defined—on the x-axis or the y-axis—you get horizontal or vertical bands. The `from` and `to` properties define the limits of the bands. Using the `color` property, you can decide with which color it will be drawn. In addition, within each band you can add a text reference thanks to the `label` property.

This example shows the use of vertical bands, so you need to add the `plotBands` component on the x-axis, as shown in Listing 18-11.

Listing 18-11. ch18_01g.html

```
xAxis: {
    ...
    plotBands: [{
       from: Date.UTC(2013, 3, 27, 8, 0, 0),
       to: Date.UTC(2013, 3, 27, 9, 0, 0),
```

```
      color: '#CACACA',
      label: {
         text: 'Close',
         style: {
            color: '#000000'
         }
      }
   },{
      from: Date.UTC(2013, 3, 27, 19, 0, 0),
      to: Date.UTC(2013, 3, 27, 20, 0, 0),
      color: '#CACACA',
      label: {
         text: 'Close',
         style: {
            color: '#000000'
         }
      }
   },{
      from: Date.UTC(2013, 3, 27, 12, 30, 0),
      to: Date.UTC(2013, 3, 27, 14, 0, 0),
      color: '#FFE7B6',
      label: {
         text: 'Lunch',
         style: {
            color: '#000000'
         }
      }
   }],
   ...
},
```

Be sure to erase the settings of the grid with alternating areas, which is no longer needed, by deleting the `alternateGridColor` property and all of its contents within the `yAxis` component. These changes are indicated by the bold code lines in Listing 18-12.

Listing 18-12. ch18_01g.html

```
yAxis: {

//  You have to delete the following rows
    alternateGridColor: {
       linearGradient: {
          x1: 0, y1: 1,
          x2: 1, y2: 1
       },
       stops : [
          [0, "#F8F8EE"],
          [1, "#A2B9A6"]
       ],
    },
//

},
```

Moreover, you can make further customizations. If you do not like the default colors, you can replace them, for example, with the sequence of colors in jqPlot. You assign them directly to the `colors` component (see Listing 18-13).

Listing 18-13. ch18_01g.html

```
$('#myChart').highcharts({
    colors: ["#4bb2c5", "#c5b47f", "#EAA228", "#579575"],
    chart: {
    ...
```

Figure 18-7 shows the result.

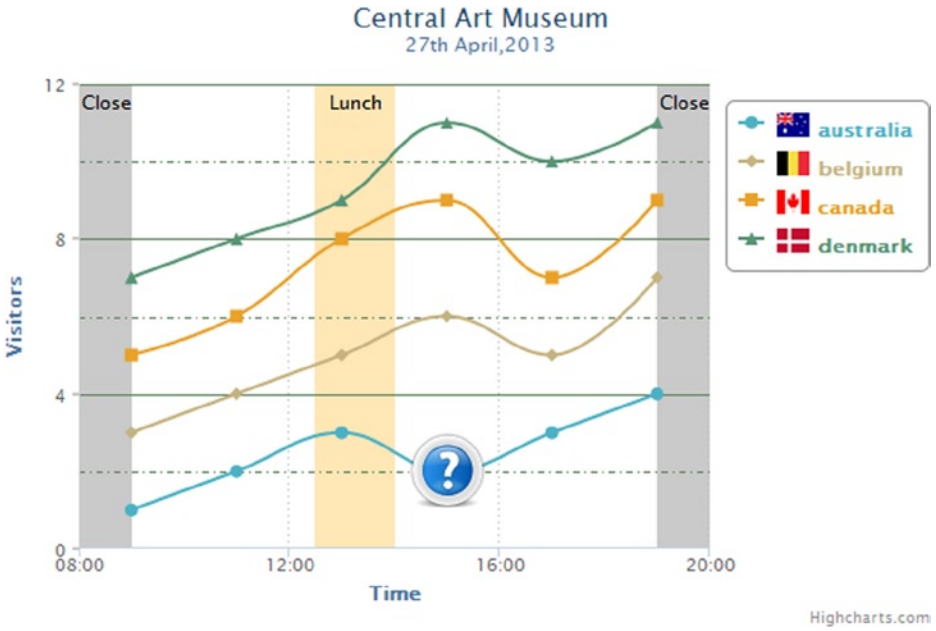


Figure 18-7. *Delimiting some areas of the chart with colored bands can enrich it with further information*

Customizing the Marker Points

For those who are used to working with jqPlot, Highcharts provides many additional features. You have seen how to customize both the legend and the tooltips. And what of the marker points? Well, you can affect many operations even on these chart components. For example, suppose you wanted to display an icon in place of one of the marker points, in order to highlight a particular characteristic of that given point on the graph at a first glance (see the note that follows the code listing). In order to do this, you specify the appropriate properties directly in the input array (see Listing 18-14), precisely at the point where you want the marker point.

Listing 18-14. ch18_01g.html

```
var series1 = [1, 2, 3, {y:2, marker:{ symbol:'url(icon/info.png)'}}, 3, 4];
var series2 = [3, 4, 5, 6, 5, 7];
var series3 = [5, 6, 8, 9, 7, 9];
var series4 = [7, 8, 9, 11, 10, 11];
```

■ **Note** You can find the icon image `info.png` in the source code accompanying the book or you can also download similar icons from the OpenIcon Library site at the following URL: `http://openiconlibrary.sourceforge.net`. This site contains a huge collection of icons available for downloading.

Each value in the input array can be treated as a component object, whereby it is possible to specify all the properties and thus to override the default values. Such an approach allows you to differentiate behavior and layout at the level of individual data points.

That is in fact what will appear in place of the marker point (see Figure 18-8).

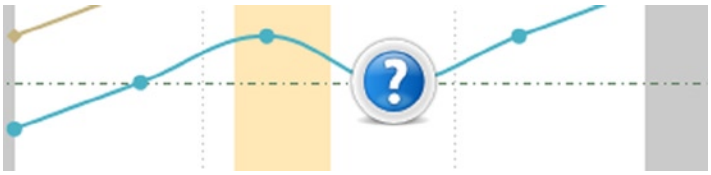


Figure 18-8. *Any icon can replace the original marker point, giving further info about a specific point in a series*

The Themes of Highcharts

In jqPlot you have seen the management of the styles of the various components through the definition of the properties of Cascading Style Sheets (CSS) classes, but you haven't yet seen themes in action. You might consider a theme such as a particular set of styles on the various constituent elements of the web page. These configurations are stored in special files so that they can be reused. They often produce a personalized style that's easily recognizable. These configurations are often referred to as **themes**.

Within its distribution, Highcharts provides some themes you can use to characterize your chart:

- Grid
- Skies
- Gray
- Dark blue
- Dark green

To set these themes on your web site, you must include the JS file that shows the same style name. For the local method, use the following:

```
<script src="../src/js/themes/dark-blue.js" type="text/javascript"></script>
```

Or if you prefer to use a CDN service:

```
<script src="http://code.highcharts.com/themes/dark-blue.js"></script>
```

The effects of these themes on the layout of your chart, shown in Figure 18-9, are remarkable.

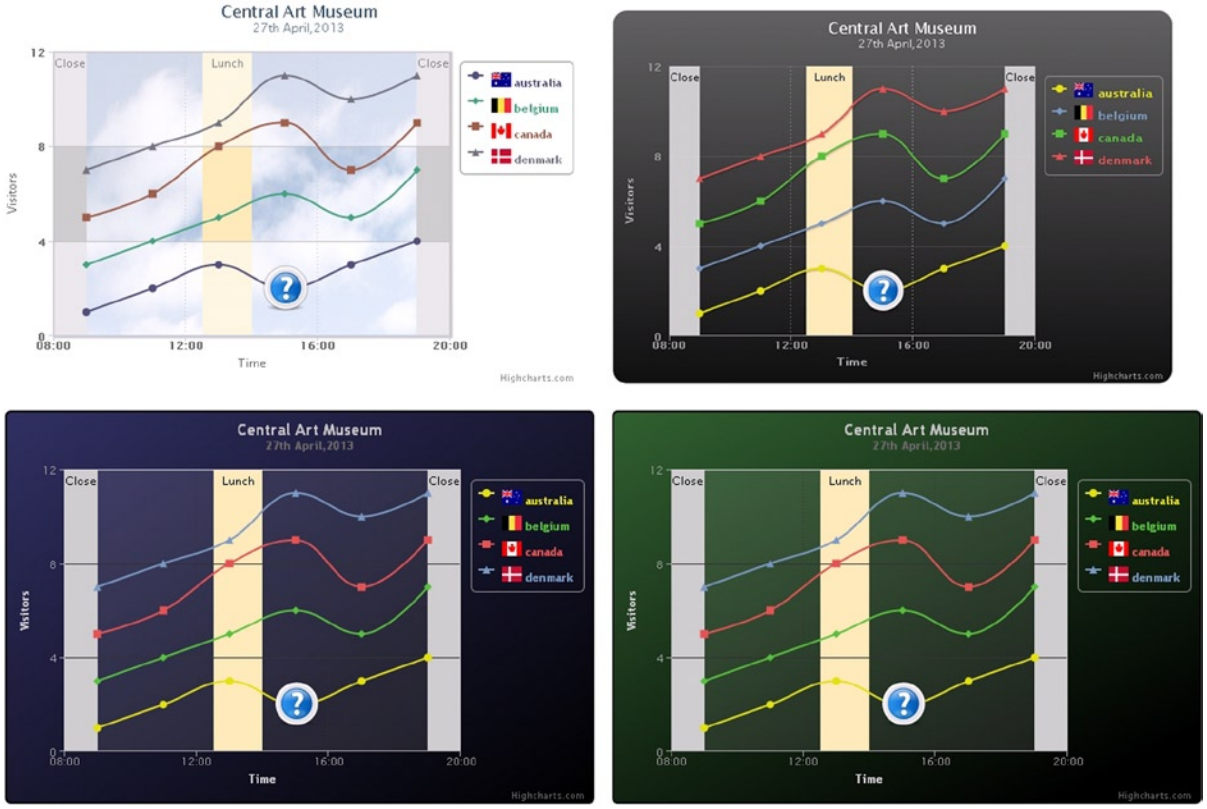


Figure 18-9. *In selecting a theme, you can accentuate certain aspects of your chart and increase the brightness of some colors*

Reading Data from a File

Often, those who use Highcharts or similar libraries need to read the data contained in a file. This is very important because your web page, with its JavaScript code, can be considered a real application that reads the data produced somewhere else, and this data can vary each time the page is called by the browser.

The application that produces the data writes the data into a file that's accessible from the network, such as a CSV file. When users want to see the data in a chart, the browser calls the HTML page where you have written the Highcharts code. It is the HTML page that reads, each time, the data contained in the CSV file (which may not even be on the same computer) so the users always see the updated data. The developer does not need to reprogram a different page each time, and the application that writes the data to a CSV file will do so independently. A further step is taken when there's a database instead of a file, from which to request the data.

Reading a CSV File Using $.get()

To read the contents of a file, jQuery library provides a very useful function: `$.get()`.

```
$.get("aFile.html", function( data ) { ... });
```

This function loads files from the servers using an HTTP `GET` request, and allows you to read its content using a nested function. (For further information, see the jQuery API reference at `http://api.jquery.com/jQuery.get/`.)

To illustrate the process of file reading, a CSV file is a good choice, since this format is quite common and simple to use. Listing 18-15 shows the process of reading a simple CSV file containing columns of data.

Listing 18-15. data_08a.csv

```
date,open,min,max,close,
08/08/2012,1.238485,1.2227,1.250245,1.2372,
08/09/2012,1.23721,1.21671,1.24873,1.229295,
08/10/2012,1.2293,1.21417,1.25168,1.228975,
08/12/2012,1.229075,1.21747,1.23921,1.22747,
08/13/2012,1.227505,1.21608,1.24737,1.23262,
08/14/2012,1.23262,1.22167,1.248555,1.232385,
08/15/2012,1.232385,1.21641,1.254355,1.228865,
08/16/2012,1.22887,1.215625,1.247305,1.23573,
08/17/2012,1.23574,1.21891,1.23824,1.2333,
08/19/2012,1.23522,1.22291,1.245275,1.23323,
08/20/2012,1.233215,1.21954,1.256885,1.2351,
08/21/2012,1.23513,1.21465,1.258785,1.247655,
08/22/2012,1.247655,1.22315,1.264415,1.25338,
08/23/2012,1.25339,1.232465,1.288965,1.255995,
08/24/2012,1.255995,1.228175,1.276665,1.2512,
08/26/2012,1.25133,1.23042,1.292415,1.25054,
08/27/2012,1.25058,1.239025,1.28356,1.25012,
08/28/2012,1.250115,1.22656,1.287695,1.2571,
08/29/2012,1.25709,1.221895,1.29736,1.253065,
08/30/2012,1.253075,1.218785,1.27639,1.25097,
08/31/2012,1.25096,1.239375,1.283785,1.25795,
09/02/2012,1.257195,1.226845,1.298705,1.257355,
09/03/2012,1.25734,1.22604,1.271095,1.258635,
09/04/2012,1.25865,1.23264,1.282795,1.25339,
09/05/2012,1.2534,1.230195,1.27245,1.26005,
09/06/2012,1.26006,1.246165,1.28513,1.26309,
09/07/2012,1.26309,1.232655,1.291765,1.281625,
09/09/2012,1.28096,1.24915,1.311295,1.279565,
09/10/2012,1.27957,1.24552,1.30036,1.27617,
09/11/2012,1.27617,1.2459,1.29712,1.28515,
09/12/2012,1.28516,1.241625,1.31368,1.290235,
09/13/2012,1.227505,1.20608,1.25737,1.23262,
09/14/2012,1.24262,1.22167,1.278555,1.232385,
09/15/2012,1.252385,1.21641,1.284355,1.228865,
09/16/2012,1.24887,1.225625,1.257305,1.23573,
09/17/2012,1.24574,1.22891,1.26824,1.2333,
09/19/2012,1.24522,1.23291,1.255275,1.23323,
09/20/2012,1.233215,1.21954,1.256885,1.2351,
09/21/2012,1.22513,1.21465,1.248785,1.247655,
09/22/2012,1.227655,1.21315,1.254415,1.25338,
09/23/2012,1.22339,1.202465,1.258965,1.255995,
09/24/2012,1.215995,1.208175,1.256665,1.2512,
09/26/2012,1.22133,1.20042,1.252415,1.25054,
09/27/2012,1.22058,1.209025,1.25356,1.25012,
09/28/2012,1.230115,1.21656,1.257695,1.2571,
09/29/2012,1.24709,1.221895,1.25736,1.253065,
```

```
09/30/2012,1.233075,1.218785,1.25639,1.25097,
09/31/2012,1.24096,1.229375,1.263785,1.25795,
10/02/2012,1.257195,1.226845,1.258705,1.257355,
10/03/2012,1.25734,1.22604,1.271095,1.258635,
10/04/2012,1.25865,1.23264,1.282795,1.25339,
10/05/2012,1.2534,1.210195,1.28245,1.26005,
10/06/2012,1.26006,1.226165,1.28513,1.26309,
10/07/2012,1.26309,1.232655,1.281765,1.281625,
10/09/2012,1.28096,1.24915,1.291295,1.279565,
10/10/2012,1.29957,1.25552,1.31036,1.27617,
10/11/2012,1.30617,1.2559,1.32712,1.28515,
10/12/2012,1.28516,1.261625,1.31368,1.290235,
```

The file contains five columns. The first is a list of `datetime` values and marks the time along the x-axis, whereas the next four are OHLC (open-high-low-close) data types. To handle the first type of data, you need to build a proper parser. This will contain a `regex` expression to capture the values of day, month, and year in the first column, and a function that reconstructs them in a format that the code can manage (see Listing 18-16).

Listing 18-16. ch18_02a.html

```
Highcharts.Data.prototype.dateFormats['m/d/Y'] = {
    regex: '^([0-9]{1,2})\/([0-9]{1,2})\/([0-9]{4})$',
    parser: function (match) {
       return Date.UTC(match[3], match[1] - 1, +match[2]);
    }
};
```

Now you must make sure that the JavaScript code can read the data contained in the CSV file. To this end, you have to declare the function `$.get()`, and the file name and a function that scan the contents passed as arguments. Within the `$.get()` function, you put the `highcharts()` function, so that it has the visibility of all data contained in the file, through the `csv` variable (see Listing 18-17).

Listing 18-17. ch18_02a.html

```
$.get('data_08a.csv', function (csv) {
    $('#myChart').highcharts(options);
});
```

Now it is time to define all the properties of the various components of the chart. As in jqPlot, even with Highcharts, when considering increasingly complex cases, it is better to reason in modules. So, from now on, you'll define the configuration object externally via the `options` variable, which you then pass as an argument to the `highcharts()` function (see Listing 18-18).

Listing 18-18. ch18_02a.html

```
$.get('data_08.csv', function (csv) {
    var options = {
       colors: ["#005B06", "#000000", "#9D3C27", "#000000"],
       data: {
          csv: csv
       },
```

```
        subtitle: {
           text: 'Prices of the day',
           style: {
              color: '#005B06',
              fontSize: '12px'
           }
        },
        title: {
           text: 'Spaghetti Lunghetti',
           style: {
              color: '#005B06',
              fontSize: '16px'
           }
        },
        xAxis: {
           type: 'datetime',
           tickInterval: 7 * 24 * 3600 * 1000, // one week
           tickWidth: 0,
           gridLineWidth: 1,
           labels: {
              align: 'left'
           }
        },
        yAxis: {
             title: {
                text: 'Dollars ($)',
                style: {
                    color: '#005B06',
                    fontSize: '12px'
                }
            }
        },
        plotOptions: {
           area: {
              fillColor: {
                 linearGradient: { x1: 0, y1: 0, x2: 0, y2: 1},
                 stops: [ [0, '#00602F'], [1, '#FFFFFF']]
              },
              lineWidth: 1,
              marker: {
                 enabled: false
              },
              shadow: false,
              states: {
                 hover: { lineWidth: 1 }
              },
              threshold: null
           },
        },
        series: [{
           name: 'Open',
           type: 'area',
```

```
            lineWidth: 4,
            marker: {
               radius: 4
            },
         },{ visible: false},
         {
            name: 'Max',
            type: 'spline',
            lineWidth: 4,
            marker: {
               enabled: false
            }
         },{ visible: false}]
   };
   $('#myChart').highcharts(options);
});
```

To activate reading the data written into a file, you need to include the `data` module, which you can find in the distribution. For the local method, use the following:

```
<script src="../src/js/modules/data.js"></script>
```

Or if you prefer to use a CDN service:

```
<script src="http://code.highcharts.com/modules/data.js"></script>
```

Loading the page in the browser, you get the chart in Figure 18-10.

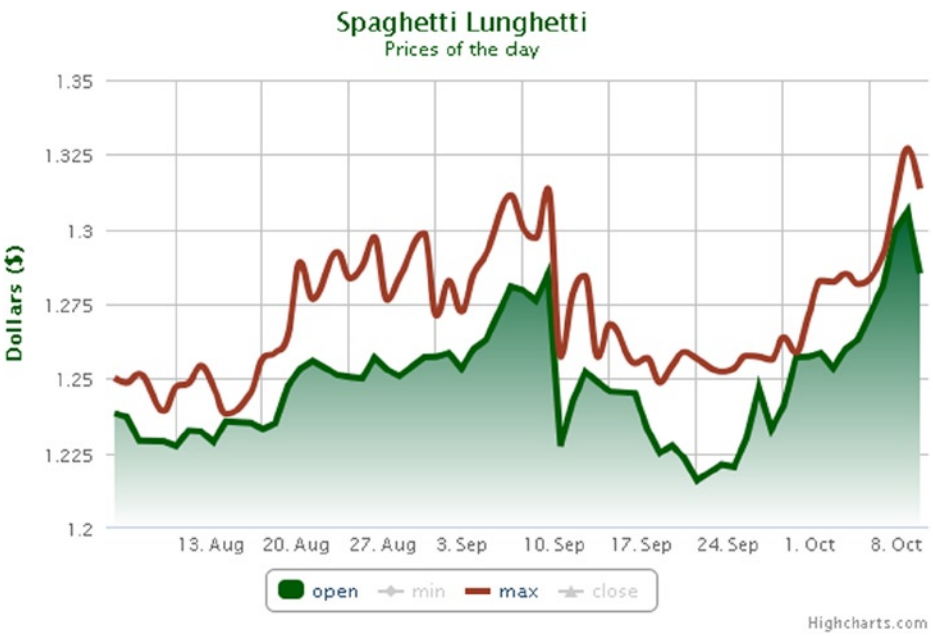


Figure 18-10. *By default all the series in an imported CSV file are read; you can hide them if need be*

Excluding CSV Columns from Your Data

Note that the CSV file contains a list of OHLC values that occupy the last four columns of data. Suppose that, in this case, you are interested in displaying only those columns containing the `open` and `max` values. You want to neglect the remaining two. If you look at the `series` object, you still need to define four different series, the second and fourth of which are hidden in the chart (not deleted!). Thus, the references of the `min` and `close` series remain in the legend. If you click on them, these series will appear on the chart, and many times this is not desirable.

You must take a different approach when you want to display only the first columns, excluding the last (if you want, for example, to display only `open`, or `open` and `min`, or `open`, `min`, and `max`). In order to do this, you need to specify only the first column, or the first two, or the first three respectively in the `series` object. But you cannot skip a series to consider the next directly.

So this method is optimal if you intend to view all of the series in the file, but sometimes that is not the case. You'll often have files with a large number of columns, but want to extract only a few. You then need to follow a different approach, for example, to enter a JavaScript function that acts as a parser and extracts only the columns that interest you. You'll now see how to modify the previous example in order to do just that.

First insert a parser immediately after the function `$.get()` has read all the data contained in the CSV file. You define two arrays that will contain the data columns `open` and `max`, both correlated to their date. Subsequently through a line-by-line scan, the two arrays will be filled with the corresponding values. In this way, you exclude the columns that do not interest you and you can work only on the `open` and `max` variables, which will be passed to the `series` object. Listing 18-19 shows the code for this.

Listing 18-19. ch18_02b.html

```
$.get('data_08a.csv', function (csv) {
    var open = [];
    var max = [];
    var lines = csv.split('\n');
    $.each(lines, function(lineNo, line) {
       var items = line.split(',');
       if (lineNo != 0) {
          open.push([items[0], parseFloat(items[1])]);
          max.push([items[0], parseFloat(items[3])]);
       }
    });
...
```

The `data` object inside `options` (see Listing 18-20) is no longer necessary; therefore, you should delete it.

Listing 18-20. ch18_02b.html

```
var options = {
    colors: ["#005B06", "#9D3C27"],
    //data: {
       //csv: csv
    //},
```

You pass the two arrays directly to the `series` object using the `data` property (see Listing 18-21).

Listing 18-21. ch18_02b.html

```
series: [{
    data: open,
    name: 'Open',
    type: 'area',
    lineWidth: 4,
    marker: {
       radius: 4
    },
},{
    data: max,
    name: 'Max',
    type: 'spline',
    lineWidth: 4,
    marker: {
       enabled: false
    }
}]
```

Figure 18-11 shows the new chart with no more references to the unwanted series.

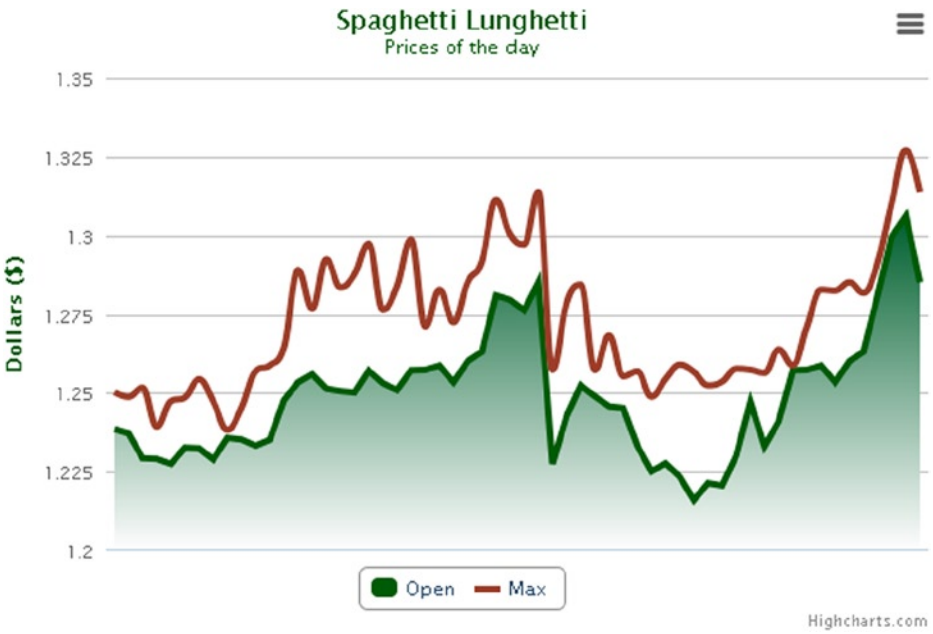


Figure 18-11. *By adding a parser, it is possible to filter only for series of interest (look at the legend)*

Exporting the Chart

A completely new feature that Highcharts offers is the capability for users to export the chart from your browser in various formats, including as a PNG or JPG image, as an SVG vector, or as a PDF document. It is even possible to print the image. All of this is done via a menu you access using the small button in the upper-right corner of the chart (see Figure 18-12).

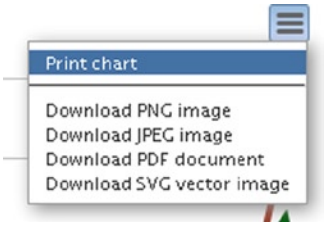


Figure 18-12. *By clicking the icon in the upper-right corner of the chart, you'll access a context menu that contains options for exporting the chart*

This button will appear in your charts whenever you include the `exporting` module in the web page. You can find this module in the Highcharts distribution. For the local method, use the following:

```
<script src="../src/js/modules/exporting.js"></script>
```

Or if you prefer to use a CDN service:

```
<script src="http://code.highcharts.com/modules/exporting.js"></script>
```

The Master Detail Chart

An example of line chart that aptly illustrates the functionality that a library such as Highcharts can provide is the **master detail chart**. This type of line chart is composed of two representations of the same line chart shown simultaneously. The purpose is to bring focus to a particular part of the line chart without losing the chart as a whole.

The two charts are called **master** and **detail** charts. As you can see in Figure 18-13, the master chart is usually much smaller and is placed in a marginal position with respect to the detail chart; it is often shown at the bottom with its x-axis aligned with the detail chart, so as to provide a sense of perspective. The detail chart is located on the foreground so that the users can focus on it. The detail chart shows in more detail the selected area of the master chart.

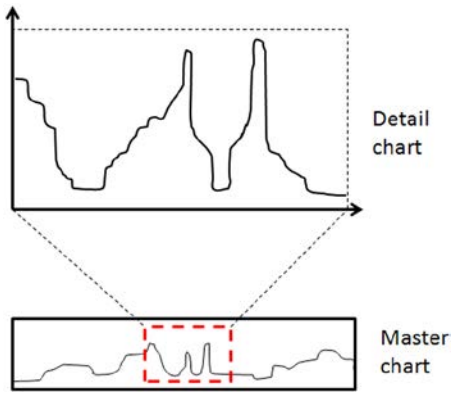


Figure 18-13. *A master and detail chart*

Master detail charts are a good choice when you have to deal with large amounts of data that would be hard to view in their entirety. Often this data is written in external files taken from other applications. Continuing with the previous Spaghetti Lunghetti prices example, you'll build a master detail chart from a CSV file.

Before starting with the real example, you need to include the library files and the dark green theme with the local method:

```
<script type="text/javascript" src="../src/js/jquery-1.9.1.js"></script>
<script type="text/javascript" src="../src/js/highcharts.js"></script>
<script type="text/javascript" src="../src/js/modules/data.js"></script>
<script type="text/javascript" src="../src/js/themes/dark-green.js"></script>
```

Or if you prefer to use a CDN service:

```
<script src="http://code.jquery.com/jquery-1.9.1.min.js"></script>
<script src="http://code.highcharts.com/highcharts.js"></script>
<script src="http://code.highcharts.com/modules/data.js"></script>
<script src="http://code.highcharts.com/themes/dark-green.js"></script>
```

Since in this example you'll be using the same CSV file from the previous example (see Listing 18-15), you'll need a proper parser to read the data correctly. This parser will contain a regex expression to capture the values of day, month, and year. Listing 18-22 present the parser that you had previously developed anew (see Listing 18-16), as it's still valid.

Listing 18-22. ch18_03.html

```
<script>
$(function () {
    Highcharts.Data.prototype.dateFormats['m/d/Y'] = {
       regex: '^([0-9]{1,2})\/([0-9]{1,2})\/([0-9]{4})$',
       parser: function (match) {
          return Date.UTC(match[3], match[1] - 1, +match[2]);
       }
   };
});
</script>
...
<div id="myChart" style="width: 600px; height: 400px;"></div>
```

Don't forget to add a `<div>` element with `myChart` as `id`.

The master detail chart is made up of two line charts, which you define through two variables: `masterChart` and `detailChart` (see Listing 18-23). These two charts will be drawn in two distinct areas, `master-container` and `detail-container`. Thanks to jQuery, you'll create two `<div>` elements with these identifiers. These two charts will be created by two JavaScript functions, `createDetail()` and `createMaster()`. This duality is maintained even in the definition of the two configuration objects that govern the layouts and behaviors of the two charts: `detailOptions` and `masterOptions`.

Listing 18-23. ch18_03.html

```
$.get('data_08a.csv', function (csv) {

    var masterChart,detailChart;
    function createMaster() {
       masterChart = $('#master-container')
       .highcharts(masterOptions, function(masterChart) {
          createDetail(masterChart);
       }).highcharts(); // return chart instance
    }
```

```
    function createDetail(masterChart) {
       var detailData = [],
       detailStart = Date.UTC(2012, 7, 1);

       jQuery.each(masterChart.series[0].data, function(i, point) {
          if (point.x >= detailStart) {
             detailData.push(point.y);
          }
       });
       detailChart = $('#detail-container').highcharts(detailOptions)
          .highcharts();
    }

    var $container = $('#myChart')
       .css('position', 'relative');

    var $detailContainer = $('<div id="detail-container">')
      .appendTo($container);

    var $masterContainer = $('<div id="master-container">')
       .css({ position: 'absolute', top: 300, height: 80, width: '100%'})
       .appendTo($container);

    createMaster();
});
```

Each time the chart is redrawn, the `createDetail()` function is called first. It creates the corresponding chart. Inside the function, there is the call to the `createMaster()` function, which creates the second chart based on the selected area in the first chart.

Now you must define two configuration objects. In Listing 18-24, you define the options regarding the master chart. When looking at the `masterOptions` object, focus your attention on the definition of the `events` property inside the `chart` object. As you can see, a generic function is defined within the `selection` property. In fact, in the master chart, you'll need to select a particular range with the mouse. This chart section will thus be displayed in the detail chart. It is precisely this feature that is implemented through this function. The `extremesObject` variable stores the interval of the x-axis covered by the selected area, passing it to the `event.xAxis[0]` value. Hence, you extract the maximum and minimum values of this range (`min` and `max`), which are just the extremes you need. These values are then used both to draw the points in the detail chart and to draw the shaded areas in the master chart. In fact, the next step is to use an `each()` function to select the data between the `min` and `max` values. All of these data points are stored into the `detailData` array. This array will be the input data array for the detail chart.

Listing 18-24. ch18_03.html

```
var masterOptions = {
    colors: ['#FFE76D'],
    data: {
       csv: csv
    },
    chart: {
       reflow: false,
       borderWidth: 0,
       backgroundColor: null,
       marginLeft: 50,
```

```
      marginRight: 20,
      zoomType: 'x',
      events: {
         selection: function(event) {
            var extremesObject = event.xAxis[0],
            min = extremesObject.min,
            max = extremesObject.max,
            detailData = [],
            xAxis = this.xAxis[0];
            jQuery.each(this.series[0].data, function(i, point) {
               if (point.x > min && point.x < max) {
                  detailData.push({
                     x: point.x,
                     y: point.y
                  });
               }
            });
            xAxis.removePlotBand('mask-before');
            xAxis.addPlotBand({
               id: 'mask-before',
               from: Date.UTC(2012, 1, 1),
               to: min,
               color: 'rgba(1, 1, 1, 0.5)'
            });
            xAxis.removePlotBand('mask-after');
            xAxis.addPlotBand({
               id: 'mask-after',
               from: max,
               to: Date.UTC(2013, 1, 1),
               color: 'rgba(1, 1, 1, 0.5)'
            });
            detailChart.series[0].setData(detailData);
            return false;
         }
      }
   },
   title: {
      text: null
   },
   xAxis: {
      type: 'datetime',
      showLastTickLabel: true,
      title: {
          text: null
      }
   },
   yAxis: {
      gridLineWidth: 0,
      labels: {
         enabled: false
      },
```

```
        title: {
            text: null
        },
        min: 1.25,
        max: 1.31
    },
    tooltip: {
        formatter: function() {
            return false;
        }
    },
    legend: {
        enabled: false
    },
    credits: {
        enabled: false
    },
    plotOptions: {
        series: {
            fillColor: {
                linearGradient: [0, 0, 0, 70],
                stops: [
                    [0, '#FFE76D'],
                    [1, 'rgba(0, 0, 0, 0)']
                ]
            },
            lineWidth: 1,
            marker: {
                enabled: false
            },
            shadow: false,
            states: {
                hover: {
                    lineWidth: 1
                }
            },
            enableMouseTracking: false
        }
    },
    series: [{
        name: 'open',
        type: 'area',
        lineWidth: 2
    }],
    exporting: {
        enabled: false
    }
};
```

Another use of `min` and `max` values is to delimit the selected area in the master chart. You achieve this by adding two bands and identifying them with the `mask-before` and `mask-after` IDs. These bands are meant to shade the area that has not been selected. Thus the first band will start from a date that will be the lowest possible value of

your experimental data (beyond the left edge of the master chart). So, for example, you assign the value `Date.UTC(2012,1,1)` to the `from` property, and the band will correspond to the `min` value that's assigned to the `to` property. The same thing applies to the second band. It starts from the `max` value, which is assigned to the `from` property, and it ends up with the highest possible value of the experimental data (beyond the right edge of the master chart). You can assign `Date.UTC(2013,1,1)` to the `to` property.

In Listing 18-25, you define the options for the detail chart. As you can see, this definition is not very different from the line chart's definition.

Listing 18-25. ch18_03.html

```
var detailOptions = {
    colors: ["#FFE76D"],
    data: {
       csv: csv
    },
    chart: {
       marginBottom: 120,
       reflow: false,
       marginLeft: 50,
       marginRight: 20,
       style: {
          position: 'absolute'
       }
    },
    credits: {
       enabled: false
    },
    title: {
       text: 'Spaghetti Lunghetti',
       style:{
          color: '#FFE76D',
          fontSize: '16px'
       }
    },
    xAxis: {
       type: 'datetime'
    },
    yAxis: {
       title: {
          text: null
       },
       maxZoom: 0.1
    },
    tooltip: {
       formatter: function() {
          var point = this.points[0];
          return Highcharts.dateFormat('%A %B %e %Y', this.x) + ':<br/><b>'+
          Highcharts.numberFormat(point.y, 2) +' USD</b>';
       },
       shared: true
    },
```

```
    legend: {
       enabled: false
    },
    plotOptions: {
       series: {
          marker: {
             enabled: false,
             states: {
                hover: {
                   enabled: true,
                   radius: 3
                }
             }
          }
       }
    },
    series: [{
       name: 'Open',
       lineWidth: 4
    }],
    exporting: {
       enabled: false
    }
};
```

Finally, here is the master detail chart (see Figure 18-14).

Figure 18-14. *The master chart is on the bottom and the detail chart is in the foreground*

From the master chart, you can use the mouse to select the section that interests you and you'll be able to see it in detail.

Bar and Pie Charts with Highcharts

In this section, you'll examine three brief examples that show how Highcharts implements the common bar and pie charts.

In fact, you'll see how simple it is to implement a bar chart. With the help of some examples, you'll see how it is even easier to switch between a grouped and a stacked mode, from horizontal to vertical. Even implementing a pie chart is very simple and intuitive. This is important because, as you'll see in the subsequent sections, bars and pies are themselves used as components for more complex types of charts.

Bar Charts

Since bar charts are simple to implement, you'll start immediately with a stacked vertical bar chart. First of all, remember to include the library files and a theme, such as the dark green theme included here:

```
<script type="text/javascript" src="../src/js/jquery-1.9.1.js"></script>
<script type="text/javascript" src="../src/js/highcharts.js"></script>
<script type="text/javascript" src="../src/js/themes/dark-green.js"></script>
```

Or if you prefer to use a CDN service:

```
<script type="text/javascript" src="http://code.jquery.com➥
    /jquery-1.9.1.min.js"></script>
<script type="text/javascript"
    src="http://code.highcharts.com/3.0.5/highcharts.js"></script>
<script type="text/javascript" src="http://code.highcharts.com/themes➥
    /dark-green.js"></script>
```

You'll see how, by just changing a property, you can get all kinds of bar charts. Then, in order to create a bar chart with vertical bars, it is necessary to specify the `type` property as `column` (see Listing 18-26).

Listing 18-26. ch18_04a.html

```
$(function () {
    var data1 = [46.6, 14.8, 0, 61.6];
    var data2 = [2.6, 13.8, 72.6, 9.1];
    var data3 = [3.3, 53.5, 77.1, 10.6];

    var options = {
       chart: {
          type: 'column'
       },
       title: {
          text: 'Nutrition label'
       },
       xAxis: {
          categories: ['Carrots', 'Beans', 'Chicken', 'Bread']
       },
       yAxis: {
          min: 0,
```

```
            title: {
               text: 'Calories'
            }
        },
        legend: {
            reversed: true
        },
        plotOptions: {
            series: {
               stacking: 'normal'
            }
        },
        series: [{
            name: 'Carbohydrate',
            data: data1
        },{
            name: 'Fat',
            data: data2
        },{
            name: 'Protein',
            data: data3
        }]
    }
    $('#myChart').highcharts(options);
});
```

And Figure 18-15 shows our chart.

Figure 18-15. *A simple stacked bar chart*

If you want a horizontal stacked bar chart, you must define the `type` property as `bar`, as shown in Listing 18-27.

Listing 18-27. ch18_04b.html

```
chart: {
    type: 'bar'
},
```

Here is the chart, converted from vertical to horizontal (see Figure 18-16).

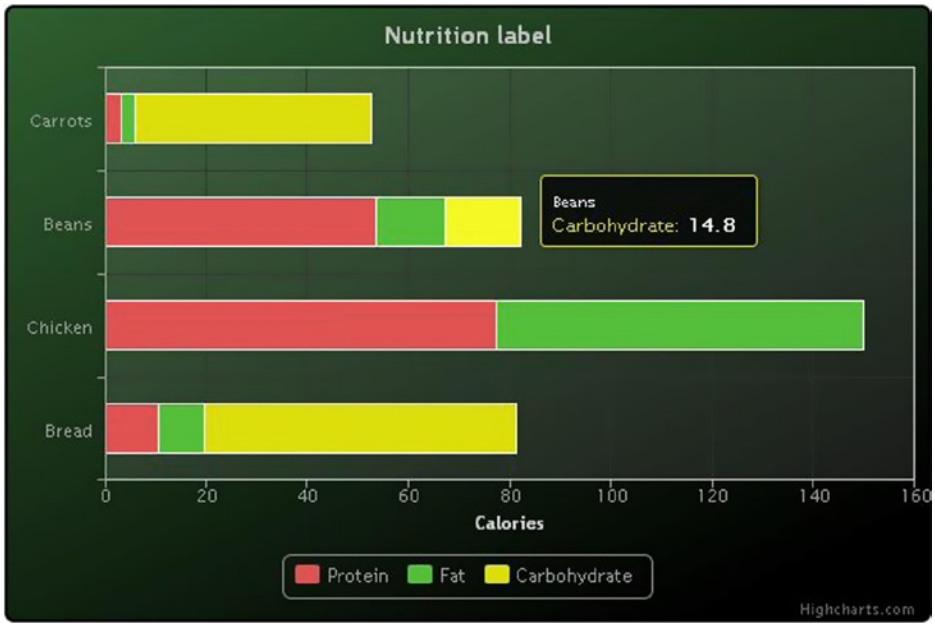


Figure 18-16. *A simple horizontal stacked bar chart*

If you instead want a normal grouped bar chart, you just delete the `plotOptions` object through `options`, as shown in Listing 18-28.

Listing 18-28. ch18_04c.html

```
//Delete the plotOptions object
plotOptions: {
    series: {
       stacking: 'normal'
    }
},
```

You'll get the two grouped bar charts, as you can see in Figure 18-17.

Figure 18-17. *The previous bar charts in the grouped version*

Pie Charts

Building a pie chart with the Highcharts library is also really simple (see Listing 18-29). Highcharts accepts the incoming data with the format [label, value], and then converts the values into percentages, from which it creates the slices in the right proportions.

Listing 18-29. ch18_04d.html

```
$(function () {
    var data2 = [['Analysis', 5], ['Designing', 10], ['Developing', 20],
                 ['Deploying', 5], ['Test', 28], ['Debugging', 23], ['Sale', 9]];
    //Add options here
    $('#myChart').highcharts(options);
});
```

Take a look at the properties specified within the `pie` object in the `plotOptions` object. By setting the `allowPointSelect` property to `true` and the `cursor` property to `pointer` (see Listing 18-30), you enable the management of a particular event. When the user clicks on a slice, it's extracted from the cake and goes to the distance specified in the `slicedOffset` property.

Listing 18-30. ch18_04d.html

```
var options = {
    colors: ['#65Af43', '#FFE76D', '#BB43F2',
             '#A50f33', '#15CACA', '#612BF3', '#FF8E04'],
    chart: {
    },
    title: {
       text: 'Developing of the X Weapon'
    },
    yAxis: {
       min: 0,
       title: {
          text: 'Calories'
       }
    },
```

```
    tooltip: {
       pointFormat: '{series.name}: <b>{point.y}%</b>',
    },
    plotOptions: {
       pie: {
          allowPointSelect: true,
          cursor: 'pointer',
          showInLegend: true,
          slicedOffset: 20
       }
    },
    series: [{
       type: 'pie',
       name: 'Budget',
       data: data2,
       borderColor: '#888888',
       borderWidth: 1,
       dataLabels: {
          enabled: true,
          color: '#bbbbbb',
          connectorColor: '#bbbbbb',
          format: '{point.name}: <b>{point.y}%</div></b>'
       },
    }]
}
```

Figure 18-18 shows the pie chart using the dark green theme.

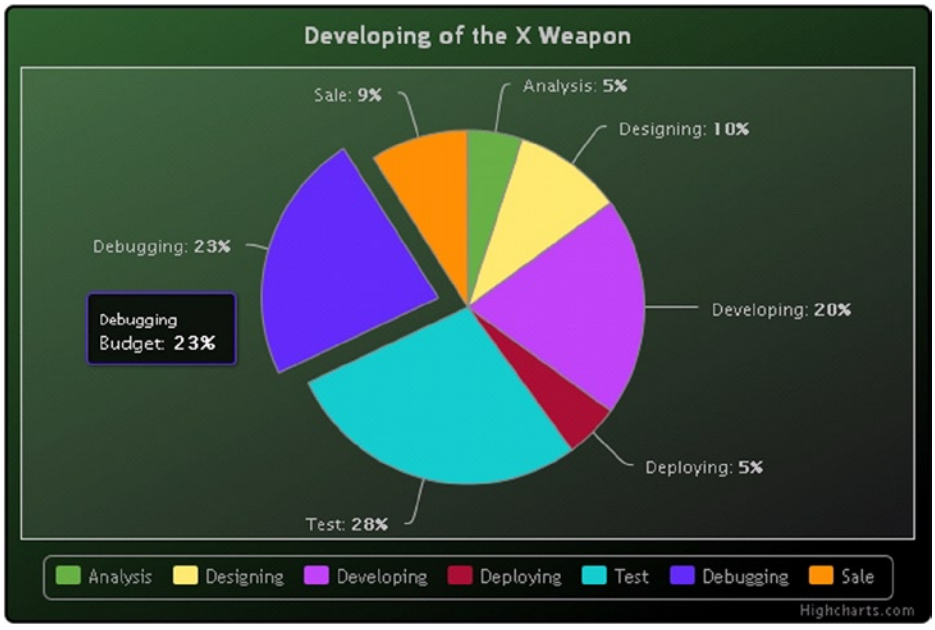


Figure 18-18. *A pie chart with all elements included*

Gantt Charts

Another type of chart that you can create easily using Highcharts is the Gantt chart (see Figure 18-19). This is a type of bar chart, developed in 1910 by Henry Gantt, used for project scheduling. In more recent times, it has been used to represent relationships between activities. For example, in books on advanced programming in Java or C++, you could commonly find that such a chart represents the synchronicity or non-synchronicity between the calls of the various functions.

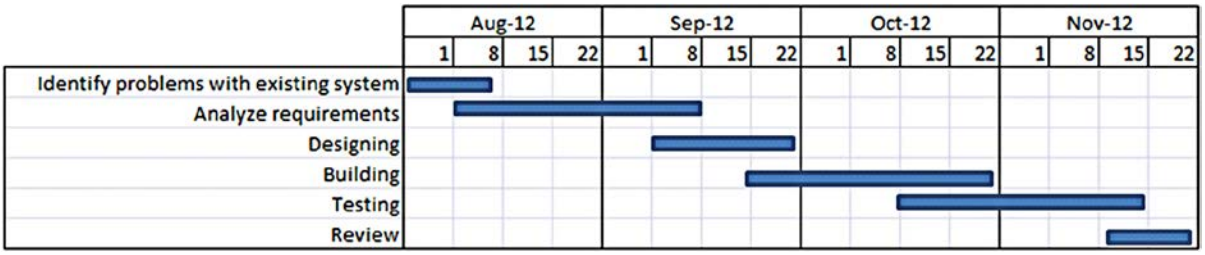


Figure 18-19. *A Gantt chart representing the development process of a project*

To implement this type of chart, you must include the extension module `highcharts-more.js`. This extension includes many types of components that have been developed more recently, including the `columrange` type, which allows you to implement the classic bars of a bar chart, but with the difference of defining both a maximum value (high) and a minimum value (low). It's therefore useful to represent a range, both horizontally (on x) and vertically (on y). The Gantt chart describes time intervals on the x-axis, whereas it requires a classification in groups along the y-axis. This classification could be easily done by passing an array of names to the `categories` property.

In the following example, you'll use the gray theme. Before you implement the chart, you need to include these two files on the web page:

```
<script type="text/javascript" src="../src/js/highcharts-more.js"></script>
<script type="text/javascript" src="../src/js/themes/gray.js"></script>
```

Or if you prefer to use a CDN service:

```
<script src="http://code.highcharts.com/highcharts-more.js"></script>
<script src="http://code.highcharts.com/themes/gray.js"></script>
```

Remember that Gantt charts express the range of times, and therefore you can define them through an array with two values for each element: [low, high]. Because they are `datetime` values, the suggested choice is to use the function `Date.UTC()` to define these intervals of time (see Listing 18-31).

Listing 18-31. ch18_05.html

```
var data1 = [ [Date.UTC(2012, 0, 1), Date.UTC(2012, 1, 15)],
              [Date.UTC(2012, 0, 20), Date.UTC(2012, 1, 28)],
              [Date.UTC(2012, 1, 4), Date.UTC(2012, 3, 30)],
              [Date.UTC(2012, 2, 10), Date.UTC(2012, 5, 15)],
              [Date.UTC(2012, 4, 1), Date.UTC(2012, 7, 19)],
```

```
            [Date.UTC(2012, 6, 1), Date.UTC(2012, 10, 15)],
            [Date.UTC(2012, 9, 1), Date.UTC(2012, 11, 28)];
```

Now you define the configuration object with the `options` variable (see Listing 18-32). In the `chart` component object, you need to set the `inverted` property to `true`. This is because you want the bars to be horizontal. You must be careful, however, because in doing so, you reverse the axes. The vertical axis will be the x-axis, and the horizontal one will be the y-axis. You need to take this into account when you define every property and especially when you need to figure out how to enter the data correctly.

Listing 18-32. ch18_05.html

```
var options = {
    chart: {
       type: 'columnrange',
       inverted: true
    },
```

The first consequence of this reversal is that you have to enter the names of the categories in the `xAxis` object in order for them to appear on the vertical axis. After you have entered a title and a subtitle in the chart, you insert an array of strings in the `categories` property containing the names to be assigned to each time interval (see Listing 18-33).

Listing 18-33. ch18_05.html

```
title: {
    text: 'Developing of the X Weapon'
},
subtitle: {
    text: 'Half Guns inc.'
},

xAxis: {
    categories: ['Analysis', 'Designing', 'Developing',
                 'Deploying', 'Test', 'Debugging', 'Sale']
},
...
```

On the other hand, you have to consider the y-axis as the time axis and define the grid accordingly. You define the axis label with the `title` property, and the type of data the axis will have to manage by specifying `datetime` in the `type` property. Then you have to set the `minPadding` and `maxPadding` properties to `0`. This will force the chart to fill the entire x-axis with the time intervals defined in the input data (see Listing 18-34).

Listing 18-34. ch18_05.html

```
yAxis: {
    title: {
       text: 'Scheduling'
    },
    type: 'datetime',
    minPadding: 0,
    maxPadding: 0,
```

```
    gridLineWidth: 1,
    gridLineColor: '#bbbbbb',
    minorTickInterval: 14 * 24 * 3600000 //2 week
},
...
```

You have already told the Highcharts library to interpret the data as `columnrange` through the `type` property within the `chart` object. Now is the time to define its properties, and you can do either within the `series` object or, as in this example, within the `columnrange` object placed under the `plotOptions` object (see Listing 18-35). Unless it is otherwise specified, you'll have all the bars of the same color because they belong to the same series, but since you want the opposite, you must activate the `colorByPoint` properties with `true` and then specify the desired color sequence as an array to the `colors` property. Finally, after disabling both the legend and the tooltips, you pass the data to the `data` property in the `series` object.

Listing 18-35. ch18_05.html

```
...
plotOptions:{
    columnrange:{
       colorByPoint: true,
       colors:['#65Af43','#FFE76D','#BB43F2','#A50f33',
          '#15CACA','#612BF3','#FF8E04']
    }
},
legend: {
    enabled: false
},
tooltip: {
    enabled: false
},
series: [{
    data: data1,
    borderColor: 'black',
    borderWidth: 2
}]
}
$('#myChart').highcharts(options);
```

At the end, here is our Gantt chart (see Figure 18-20).

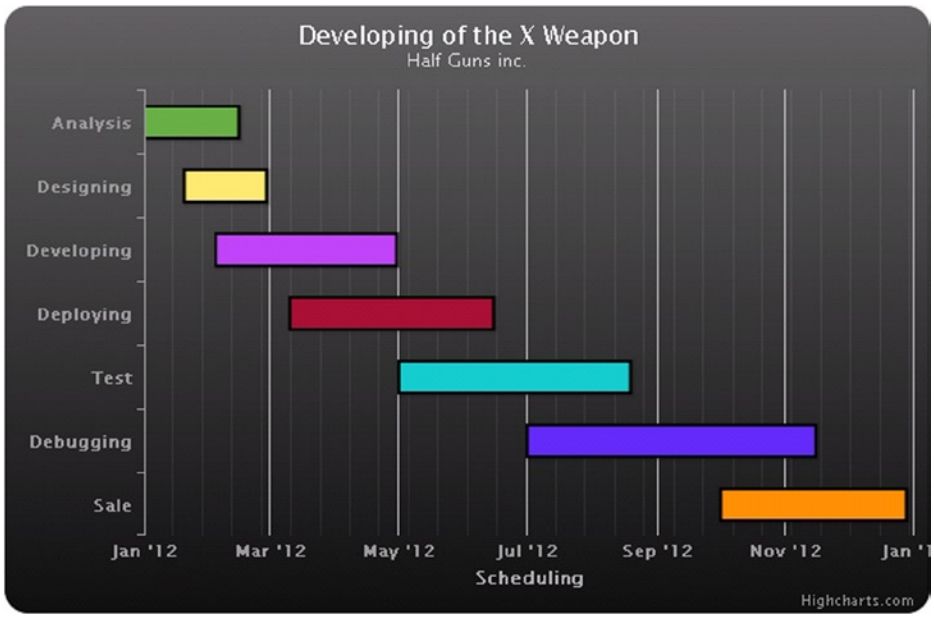


Figure 18-20. *A Gantt chart*

Combined Charts

Perhaps the aspect that distinguishes the Highcharts library and currently constitutes its showpiece is its ability to overlap several types of charts together in the same chart, thus creating truly spectacular representations. Furthermore, the library is structured in a way that the superimposed charts remain distinct in their configuration values, and thus easily manageable. You can develop them separately and then eventually bring them together.

In the next example, you'll join the pie and Gantt charts you already implemented and add a third chart, a line chart. First, import all the necessary files, including the gray theme:

```
<script type="text/javascript" src="../src/js/jquery-1.9.1.js"></script>
<script type="text/javascript" src="../src/js/highcharts.js"></script>
<script type="text/javascript" src="../src/js/highcharts-more.js"></script>
<script type="text/javascript" src="../src/js/themes/gray.js"></script>
```

Or if you prefer to use CDN service:

```
<script src="http://code.highcharts.com/jquery-1.9.1.js"></script>
<script src="http://code.highcharts.com/highcharts.js"></script>
<script src="http://code.highcharts.com/highcharts-more.js"></script>
<script src="http://code.highcharts.com/themes/gray.js"></script>
```

You need to define the data that will be used for all three charts (see Listing 18-36). In `data1`, you enter the data [low, high] defining the intervals of the Gantt chart. In `data2`, you enter the data for the Pie chart. Note that, for the sake of convenience, the values are already expressed as percentages. In `data3`, you enter the [x, y] data representing the various points of a line chart (here you defined the `datetime` values as y values and numerical values as x values, because you have reversed the axes). Finally, you add the definition of a color gradient within the `gradient` variable to avoid writing it several times in the configuration object.

Listing 18-36. ch18_06.html

```
var data1 = [[Date.UTC(2012, 0, 1), Date.UTC(2012, 1, 15)],
             [Date.UTC(2012, 0, 20), Date.UTC(2012, 1, 28)],
             [Date.UTC(2012, 1, 4), Date.UTC(2012, 3, 30)],
             [Date.UTC(2012, 2, 10), Date.UTC(2012, 5, 15)],
             [Date.UTC(2012, 4, 1), Date.UTC(2012, 7, 19)],
             [Date.UTC(2012, 6, 1), Date.UTC(2012, 10, 15)],
             [Date.UTC(2012, 9, 1), Date.UTC(2012, 11, 28)]];
var data2 = [5, 10, 20, 5, 28, 23, 9];
var data3 = [[140, Date.UTC(2012,0,1)],
             [120, Date.UTC(2012, 1, 28)],
             [58, Date.UTC(2012, 3, 30)],
             [78, Date.UTC(2012, 5, 15)],
             [44, Date.UTC(2012, 7, 19)],
             [33, Date.UTC(2012, 10, 15)],
             [1, Date.UTC(2012, 11, 28)]];
var gradient = {x1:0, y1:0, x2:0, y2:1};
```

You can now begin to define the configuration object (see Listing 18-37). Since the color sequence of the previous example worked well, you'll use it again in this example. Then you activate the `inverted` property using `true` to reverse the x- and y-axes. Finally, add a title and a subtitle to your chart.

Listing 18-37. ch18_06.html

```
var options = {
    colors:['#65Af43', '#FFE76D', '#BB43F2',
            '#A50f33', '#15CACA', '#612BF3', '#FF8E04'],
    chart: {
       inverted: true
    },
    title: {
       text: 'Developing of the X Weapon'
    },
    subtitle:{
       text: 'Half Guns inc.'
    },
```

For the combination chart, you need to define two different scales on the x-axis—one that will contain the categories as in the previous example and the other that's a linear type to represent the linear chart (see Listing 18-38). Regarding the y-axis, you have to define a time scale and also manage the grid. The main grid that follows the tick on the y-axis scans the months, whereas you'll set the minor grid to scan the two-week periods.

Listing 18-38. ch18_06.html

```
xAxis:[ {
    categories: ['Analysis', 'Designing', 'Casting',
                 'Develop', 'Test', 'Debugging', 'Sale']
},{
    title: {
       text: 'Budget'
    },
```

```
    labels: {
       enabled: false
    },
    opposite: true,
    tickInterval: 20
}],
yAxis: {
    title: {
       text: 'Scheduling'
    },
    type: 'datetime',
    minPadding: 0,
    maxPadding: 0,
    gridLineWidth: 1,
    gridLineColor: '#bbbbbb',
    minorTickInterval: 14 * 24 * 3600000 //2 week
},
```

For the Gantt chart, you need to make some changes compared to the previous version (see Listing 18-39). You apply a color gradient to the bars and the areas. Here, you can use the gradient transparency to create a dynamic appearance.

Listing 18-39. ch18_06.html

```
plotOptions:{
    columnrange:{
       colorByPoint: true,
       colors: [{
          linearGradient: gradient,
          stops: [[0, 'rgba(101, 175, 67, 1)'],
             [1, 'rgba(101, 175, 67, 0)']]
       },{
          linearGradient: gradient,
          stops: [[0, 'rgba(255, 231, 109, 1)'],
             [1, 'rgba(255, 231, 109, 0)']]
       },{
          linearGradient: gradient,
          stops: [[0, 'rgba(187, 67, 242, 1)'],
             [1, 'rgba(187, 67, 242, 0)']]
       },{
          linearGradient: gradient,
          stops: [[0, 'rgba(165, 15, 51, 1)'],
           [1, 'rgba(165, 15, 51, 0)']]
       },{
          linearGradient: gradient,
          stops: [[0, 'rgba(21, 202, 202, 1)'],
              [1, 'rgba(21, 202, 202, 0)']]
       },{
          linearGradient: gradient,
          stops: [[0, 'rgba(97, 43, 243, 1)'],
             [1,'rgba(97, 43, 243, 0)']]
       },{
```

```
            linearGradient: gradient,
            stops: [[0, 'rgba(255, 142, 4, 1)'],
               [1, 'rgba(255, 142, 4, 0)']]
         }]
      }
},
```

You also disable the legend and tooltips, and you begin to enter the properties for the three types of chart. Inside the `series` object, you define three different objects, each corresponding to a chart. When you work with combined charts, you have to keep the order in which they are drawn in mind. In fact, each chart, defined in the `series` object, is drawn over the previous one. If there are overlapping parts, they will be covered. So the order in which you define the charts into the `series` object is important.

First, define the line chart in order to make it the background. In this example, you can also modify the shape, size, and also the color of the marker. Then you define the Gantt chart, remembering to add the value `0` to the `xAxis` property. This means that the data in the Gantt chart, namely `data1`, is passed to the first x-axis. Finally, you define the properties of the pie chart, and with the `center` property you can move the pie in such a position as not to overlap the other charts (see Listing 18-40). With the `size` property, you can reduce the size of the pie in order to better adapt it to your needs. Highcharts, unlike jqPlot, includes **connectors,** which are lines connecting the label to the corresponding slice.

Listing 18-40. ch18_06.html

```
    legend: {
       enabled: false
    },
    tooltip: {
       enabled: false
    },
    series: [{
       type: 'line',
       data: data3,
       xAxis:1,
       color: ['#ddddaa'],
       lineWidth: 4,
       dashStyle: 'dash',
       marker: {
          fillColor: 'rgba(0,0,0,1)',
          lineWidth: 4,
          lineColor: '#ddddaa',
          radius: 7
       }
    },{
       type: 'columnrange',
       data: data1,
       borderWidth: 0,
       xAxis:0
    },{
       type: 'pie',
       name: 'Budget',
       data: data2,
       borderColor: '#888888',
       borderWidth: 1,
```

```
        center: [370, 40],
        size: 100,
        showInLegend: false,
        dataLabels: {
           enabled: true,
           color: '#bbbbbb',
           connectorColor: '#bbbbbb',
           formatter: function() {
              return '<b>'+ Math.round(this.percentage) +'%</b>';
           }
        },
        xAxis:1
    }]
}
$('#myChart').highcharts(options);
```

The finished combined chart is shown in Figure 18-21.

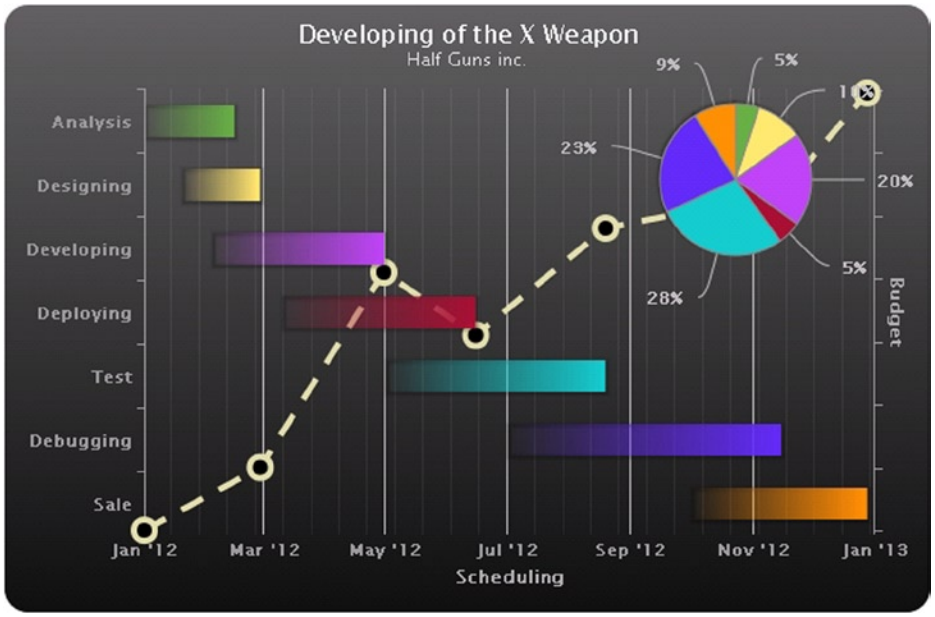


Figure 18-21. *A combined chart with three different types of chart*

Highstock Library

When you visit the official web site of Highcharts, you will find, along with this library, indications to another library called **Highstock**. This library is fully integrated into Highcharts and provides a range of features and tools to be further introduced in your charts. It allows you to create very advanced navigation tools that transform your charts into real tools for the professional analysis of large amounts of data covering very long periods of time.

If your interest in the development of the chart lead you to face very sophisticated professional needs, especially in economic analysis, this library will prove to be an indispensable tool. Since its contents primarily include tools to integrate within existing charts and their respective high specificity to particular needs, this topic is not discussed in this book. However, I strongly recommend you have a look at its contents through the various demos made available on the official web site (www.highcharts.com/products/highstock). There is also a large amount of documentation that exists in this regard (see the API Highstock Reference at http://api.highcharts.com/highstock).

Summary

This chapter, covering the **Highcharts** library, concludes the second part of this book. You have seen how this library has somehow inherited all the basic features of jqPlot and has extended them to a much more professional level.

By implementing the basis of a simple line chart, this chapter compared the two libraries, highlighting the similarities and differences. Then you saw how to handle customizations of components with Highcharts on and how to read data from a file.

With Highcharts, you have therefore implemented other types of charts such as the **Gantt chart** and the **Master and Detail chart**, both of which highlight the increased possibilities of development this library has to offer.

Finally, again taking bar charts and pie charts as examples, you learned how Highcharts allows you to combine several different types of charts simultaneously in the same drawing area, further expanding the possibility to develop even more innovative and eye-catching charts.

The next chapter begins the third and final part of the book by talking about the **D3 library**. It introduces all the basic concepts of the library and proposes some basic examples to get started. You'll discover it as an alternative to the worlds of jqPlot, Highcharts, and other similar libraries, based on jQuery and on the Canvas. You'll see how this library uses SVG technology to create all the graphic elements needed to build charts as if they were small bricks. Finally, you'll analyze the strengths and differences of everything you have seen so far.

CHAPTER 19

Working with D3

This chapter begins the third part of the book, concerning the D3 library. This library has a separate section of the book dedicated to it because it differs in many aspects from the jqPlot and Highcharts libraries. In the various sections of this chapter, and as you delve deeper into the aspects of the library in the next chapters, you'll be able to appreciate that D3 has a unique and innovative structure. First of all, it does not use jQuery, but it reproduces all the features necessary for data visualization. Whereas in the jqPlot and Highcharts libraries, chart components are already created, requiring the users only to adjust their properties via the options object, D3 has virtually the opposite approach.

The D3 library allows you to build any representation, starting with the most basic graphical elements such as circles, lines, squares, and so on. Certainly, such an approach greatly complicates the implementation of a chart, but at the same time, it allows you to develop completely new graphical representations, free from having to follow the preset patterns that the other graphic libraries provide.

Thus, in the course of this chapter, you'll become acquainted with the basic concepts that underlie this library. You'll see how some of them—such as selections, selectors, and method chains—are taken from the jQuery library. You'll also find out how to manipulate the various Document Object Model (DOM) elements, especially the creation of Scalable Vector Graphics (SVG) elements, which are the essential building blocks of the graphical representations.

The chapter closes with a brief introduction to the transformations and transitions of SVG elements.

You'll start with an introduction to this wonderful library.

FIREBUG: DEBUGGING D3 CODE

Before beginning with some practical examples, I would like to remind you to use FireBug for debugging. At the least, be sure to get a good debugging tool in JavaScript that allows you to view the DOM tree of the web page upon which you'll be working (see the "FireBug and DevTool" section in Chapter 1).

Using a debugging tool with the D3 library is essential, given that unlike the other libraries you have seen, it is not structured with premodeled objects. With D3, it is necessary to start from scratch, implementing all the chart elements one by one. Therefore, those who are familiar with development will realize that choosing a good debugging tool is essential to solving any problems that arise.

With FireBug it is possible to edit, debug, and monitor CSS, SVG, and HTML. You can change their values in real time and see the effects. It also provides a console where you can read out the log, which is suitably placed within the JavaScript code to monitor the content of the variables used. This can be achieved by calling the `log()` function of the console object and passing the variable interested as argument:

```
console.log (variable);
```

It is possible to add some text for reference, as well:

```
console.log ("this is the value:");
```

You will see that, when working with D3, FireBug is crucial for inspecting the dynamic structures of SVG elements that JavaScript generates in the DOM.

Introducing D3

D3 is a JavaScript library that, in a manner similar to the jQuery library, allows direct inspection and manipulation of the DOM, but is intended solely for data visualization. It really does its job excellently. In fact, the name D3 is derived from *data-driven documents*. D3 was developed by Mike Bostock, the creator of the Protovis library, which D3 is designed to replace.

This library is proving to be very versatile and powerful, thanks to the technologies upon which it is based: JavaScript, SVG, and CSS. D3 combines powerful visualization components with a data-driven approach to DOM manipulation. In so doing, D3 takes full advantage of the capabilities of the modern browser.

D3 allows you to bind arbitrary data to the DOM. Its strength is its capability to affect several transformations of the document. For example, a set of data could be converted into an interactive SVG graphical structure such as a chart.

You have seen that the strength of jqPlot, as a JavaScript framework, is that it provides structured solutions, which you manipulate through the settings of options. Unlike jqPlot, the strength of D3 is precisely the opposite. It provides the building blocks and tools to assemble structures based on SVG. The result of this approach is the continuous development of new structures, which are graphically rich and open to all sorts of interactions and animations. D3 is the perfect tool for those who want to develop new graphics solutions for aspects not covered by existing frameworks.

D3 does not use the jQuery library, but it has many similar concepts in it, including the method-chaining paradigm and the selections. It provides a jQuery-like interface to the DOM, which means you don't need to know all the features of SVG in much detail. In order to handle the D3 code, you need to be able to use objects and functions and to understand the basics of SVG and CSS, which are used extensively. The sacrifices that go into mastering all of this knowledge are rewarded with the amazing visualizations you can create.

SVG provides the building blocks for the artwork; it allows you to draw all the basic shape primitives such as lines, rectangles, and circles, as well as text. It allows you to build complex shapes with paths.

Starting with a Blank HTML Page

It's time to practice the concepts just outlined. First, start with a blank page, shown in Listing 19-1. This will be the starting point for all of the D3 examples.

Listing 19-1. ch19_01a.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<script src="http://d3js.org/d3.v3.js"></script>
<style>

    // CSS Style here

</style>
</head>
<body>
```

```
    <!-- HTML elements here -->

<script type="text/javascript">

    // D3 code here

</script>
</body>
</html>
```

Although, at first glance, you see only a simple HTML blank page, there are some small measures you must take when you work with D3. The most simple and clear measure is to include the library D3:

```
<script src="../src/d3.v3.js"></script>
```

Or if you prefer to use a content delivery network (CDN) service:

```
<script src="http://d3js.org/d3.v3.js"></script>
```

When entering the URL of the remote D3 library, make sure that the website always includes the latest version. Another measure, which is less obvious, is to add the `<head>` of the page:

```
<meta charset="utf-8">
```

If you do not specify this row, you will soon find out that the D3 code you added does not run. Last, but not least, where you add the various parts of the code is very important. It is advisable to include all the JavaScript code of D3 at the end of the `<body>` section, after all the HTML elements.

Using Selections and Operators

To start working with D3, it is necessary to become familiar with the concept of *selections*. Having to deal with selections involves the use of three basic objects:

- **Selections**
- **Selectors**
- **Operators**

A **selection** is an array of node elements extracted from the current document. In order to extract a specific set of elements (selection), you need to use **selectors**. These are patterns that match elements in the tree structure of the document. Once you get a selection, you might wish to perform some operations on it and so you use **operators**. As a result of their operation, you get a new selection, and so it is possible to apply another operator, and so on.

Selectors and operators are defined by the W3C (World Wide Web Consortium) APIs and are supported by all modern browsers. Generally, you'll operate on HTML documents, and so you'll work on the selection of HTML elements.

Selections and Selectors

To extract a selection from a document, D3 provides two methods:

- `select`
- `selectAll`

`d3.select("selector")` selects the first element that matches the selector, returning a selection with only one element.

`d3.selectAll("selector")` instead selects all elements that match the selector, returning a selection with all these elements.

There is no better way to understand these concepts than to do so gradually, with some simple examples. Starting from the HTML page just described, add two paragraphs containing some text and then make a selection with D3 (see Listing 19-2).

Listing 19-2. ch19_01a.html

```
<body>
<p>First paragraph</p>
<p>Second paragraph</p>
<script type="text/javascript">
    var selection = d3.select("p");
    console.log(selection);
</script>
</body>
```

`d3.select` is the top-level operator; `"p"` is the selector; and the `selection` is the returned value of the operator you assign to a variable. With this D3 command, you want to select the first element `<p>` in the web page. Using the `log` function, you can see the selection with FireBug in Figure 19-1.

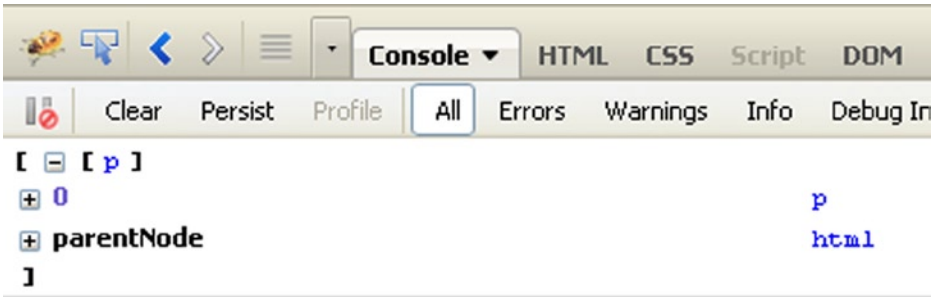


Figure 19-1. *The FireBug console enables you to see the content of the selection*

Since you used the `select()` method, you have a selection with only one element, although in the web page there are two. If you want to select both, you use `selectAll()`, as in Listing 19-3.

Listing 19-3. ch19_01b.html

```
<script type="text/javascript">
    var selection = d3.selectAll("p");
    console.log(selection);
</script>
```

Figure 19-2 shows both elements.

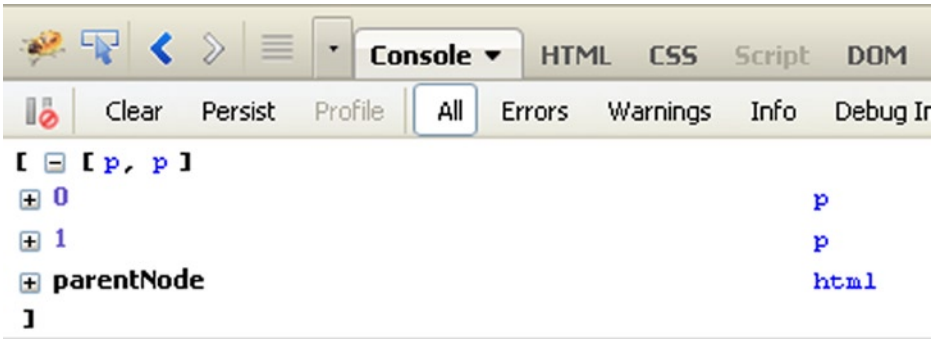


Figure 19-2. *FireBug shows the selection of all the <p> elements in the web page*

Now you have a selection with two elements. The great innovation that jQuery and D3 introduce with the concept of selection is that `for` loops are no long necessary. Instead of coding recursive functions to modify elements, you can operate on entire selections at once.

Operators

Once you have learned to make selections, it is time to apply operators to them.

An operator is a method that's applied to a selection, or generally to a set of elements, and it specifically "operates" a manipulation. For example, it can get or set a property of the elements in the selection, or can act in some way on their content. For example, you may want to replace existing text with new text. For this purpose, you use the `text()` operator, shown in Listing 19-4.

Listing 19-4. ch19_02.html

```
<body>
<p>First paragraph</p>
<p>Second paragraph</p>
<script type="text/javascript">
    var selection = d3.selectAll("p");
    selection.text("we add this new text");
</script>
</body>
```

The page now reports twice for the same text, where before there were two paragraphs (see Figure 19-3).

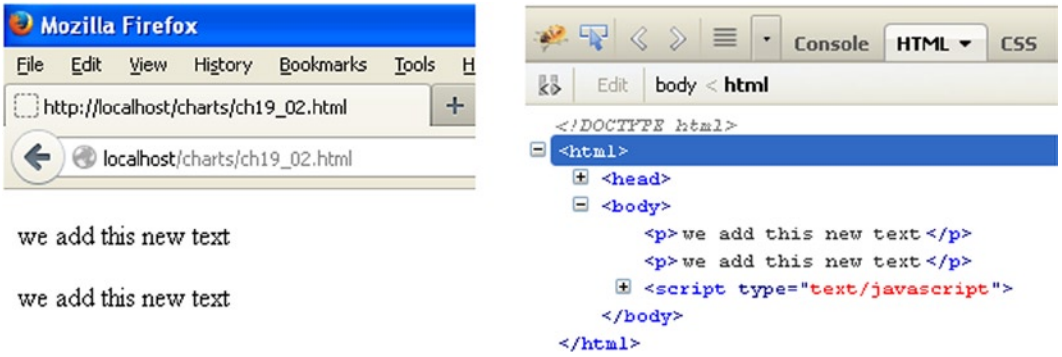


Figure 19-3. *The text contained in the two* `<p>` *elements has been replaced in the browser on the left and is shown in FireBug on the right*

You defined the variable selection and then applied the operator to this variable. But there is another way to write all this; you can use the methods of chain functionality, especially when you apply multiple operators to the same selection.

```
d3.selectAll("p").text("we add this new text");
```

You have seen that by passing a parameter to the `text()` operator, you are going to replace the existing text. So it is as if the function were `setText("new text")`. But you do not always want that. If you do not pass any arguments, the function will have a different behavior. It will return the value of the text already present. This can be very useful for further processing, or for assigning this string value to a variable or an array. Therefore, without parameters, it is as if it were `getText()`.

```
var text = d3.select("p").text();
console.log(text);
```

The text variable contains the `"First paragraph"` string (see Figure 19-4).

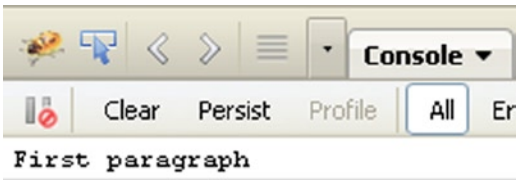


Figure 19-4. *The FireBug console shows the text contained in the selection*

There are operators for every kind of object upon which you'd want operate. These operators can set the content of:

- **Attributes**
- **Styles**
- **Properties**
- **HTML**
- **Text**

You just saw the `text()` operator in action. Next, you'll see some of the other operators.

■ **Note** If you want to learn more about operators, I suggest you visit the API reference for the D3 library at this link: https://github.com/mbostock/d3/wiki/API-Reference.

For example, it is helpful to be able to change a CSS style and you can do so with the `style()` operator. Listing 19-5 replaces the existing text using `text()` and then modifies its style to be written in red, adding the `style()` operator to the methods chain.

Listing 19-5. ch19_03.html

```
<body>
<p>Existing black text</p>
<script type="text/javascript">
    d3.selectAll("p").style('color','red').text("New red text");
</script>
</body>
```

Figure 19-5 shows the original text on the left and the newly styled text on the right.

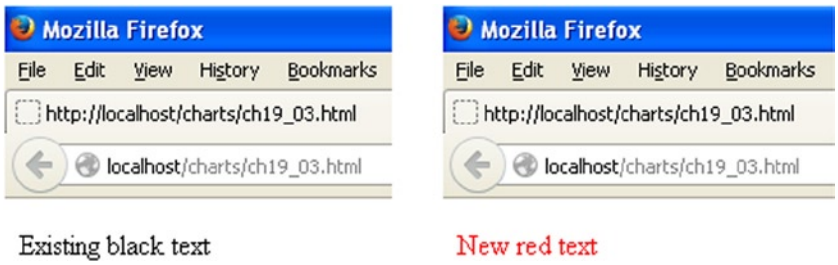


Figure 19-5. *The original text is replaced by the new red text, applying the chain method upon the selection*

Another operator, `attr()`, acts at the level of attributes of elements. This operator will be used when you create new SVG elements; in fact, it allows you to define the attributes while you are creating the tags, before inserting them in the web page. Here you can see how it can modify an existing attribute. In Listing 19-6, you're changing the alignment of a title to be displayed in the middle of the page (see Figure 19-6).

Listing 19-6. ch19_04.html

```
<body>
<h1>Title</h1>
<script type="text/javascript">
    d3.select('h1').attr('align','center');
</script>
</body>
```

Figure 19-6. *With the D3 library it is possible to dynamically add a title to a web page*

Creating New Elements

Now that you have seen how to act at the level of elements and how to modify both attributes and content, it is time to see how to create new items. To do this, D3 provides a number of operators (https://github.com/mbostock/d3/wiki/API-Reference), among which the most commonly used are:

- `html()`
- `append()`
- `insert()`

The html() Method

This section shows how the `html()` method operates. You always start from a selection and then apply this operator to add an element inside. For example, you select a particular tag as a container, and then write a string that is passed as an argument. The string then becomes the content of the tag (see Listing 19-7).

Listing 19-7. ch19_05.html

```
<body>
<p>A paragraph</p>
<script type="text/javascript">
    d3.select('p').html("<h1>New Paragraph</h1>");
</script>
</body>
```

Here, you first select the `<p>` tag with `select()` and then with `html()` you replace its contents with a new element, `<h1>`. Figure 19-7 shows the original text on the left and the newly formatted version on the right.

Figure 19-7. *The text in a paragraph element* `<p>` *is replaced with a heading element* `<h>`

You can see this change better, using FireBug (see Figure 19-8)

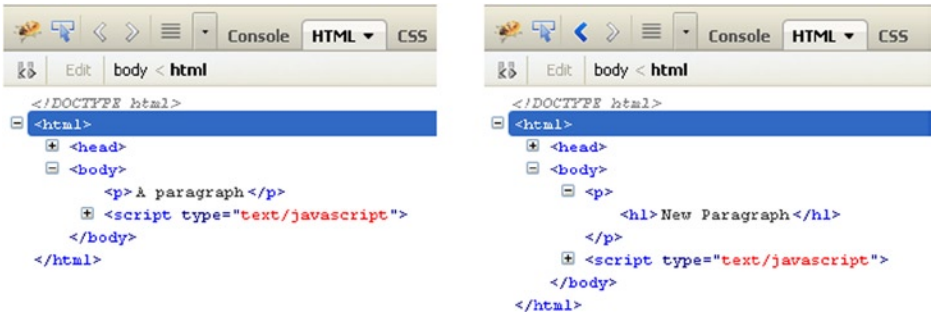


Figure 19-8. *FireBug clearly shows the insertion of the head element (on the right) to replace the content of the paragraph element (on the left)*

Practically, the `html()` function replaces the contents of the selection with the HTML code passed as an argument. Exactly as its name suggests, this function allows you to dynamically write HTML code within the elements of the selection.

The append() Method

Another popular method for adding elements is `append()`.

Recall that when you're using the `html()` operator, the content of the selected tag, if any, is replaced with the new one passed as an argument. The `append()` operator instead adds a new element, passed as its argument, to the end of all the existing elements contained in the selected tag. The content of the newly created element must be added to the chain of methods, using `text()` if it is only a string, or `append()`, `html()` or `insert()` if it is a further element.

In order to understand this last point, add an unordered list `<ul>` with some items containing fruit names to the page (see Figure 19-9).

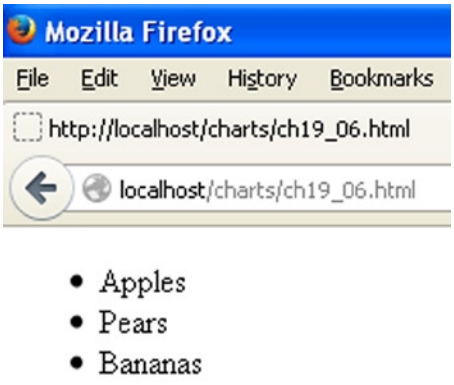


Figure 19-9. *An unordered list of three fruits*

Say that you now want to add Oranges to this list. In order to do this, you must select the unordered list tag `<ul>` and then use `append()` to add a list item tag `<li>`. But `append()` creates only the tag, so in order to insert the string `"Oranges"` inside it, you need to add the `text()` operator to the chain of methods (see Listing 19-8).

Listing 19-8. ch19_06.html

```
<body>
<ul>
    <li>Apples</li>
    <li>Pears</li>
    <li>Bananas</li>
</ul>
<script type="text/javascript">
    d3.select('ul').append('li').text("Oranges");
</script>
</body>
```

Figure 19-10 shows the list with the added element.

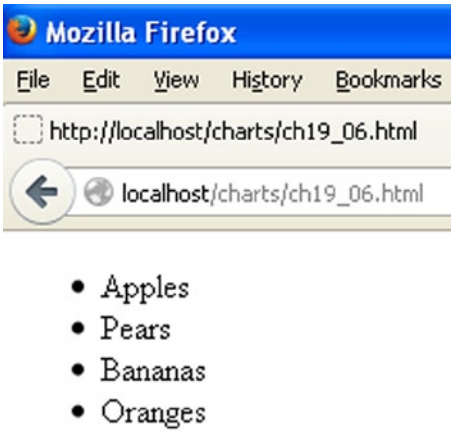


Figure 19-10. *Using the append() operator, you have added the Oranges item to the end of the list*

Figure 19-11 shows it in FireBug.

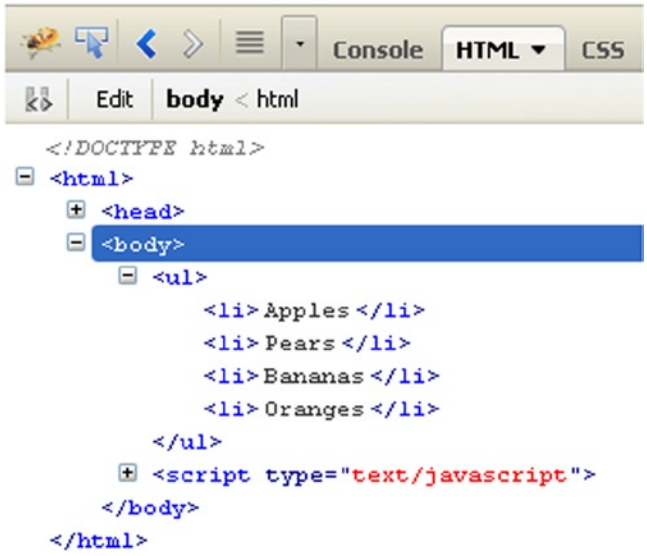


Figure 19-11. *FireBug shows the HTML structure with the added <li> element*

In this case, you have used simple text as the content for the new element added to the list, but the append() operator can do more. In fact, as previously noted, the content of an element may be yet another element. This allows you to create an entire tree of HTML elements, all by exploiting a chain method. In fact, the content of the new element created by the append() operator can in turn be created by another operator, such as another append() operator. Look at Listing 19-9. It is a simple example that will help you better understand this concept.

This time, you want to create a sub-category of fruits, Citrus fruits, in which we shall assign the Oranges, Lemons, and Grapefruits items. In order to do this, you need to add a new list item <li> with the string "Citrus fruits" as its content. This works the same way as in the previous example, concatenating the text() operator just after the append() operator. Then you need to create a new list item. This time, its content is an unordered list. Thus, you need to concatenate two append() operators in order to create a list item <li> element nested in an unordered list <ul> element. You can then add two other new elements to the nested unordered list, again with the append() operator.

Listing 19-9. ch19_06b.html

```
<body>
<ul>
    <li>Apples</li>
    <li>Pears</li>
    <li>Bananas</li>
</ul>
<script type="text/javascript">
    d3.select('ul').append('li').text("Citrus fruits");
    d3.select('ul').append('ul').append('li').text("Oranges");
    d3.select('ul').select('ul').append('li').text("Lemons");
    d3.select('ul').select('ul').append('li').text("Grapefruits");
</script>
</body>
```

Figure 19-12 shows the new nested list of citrus fruits on the browser and the HTML structure that generates it on FireBug.

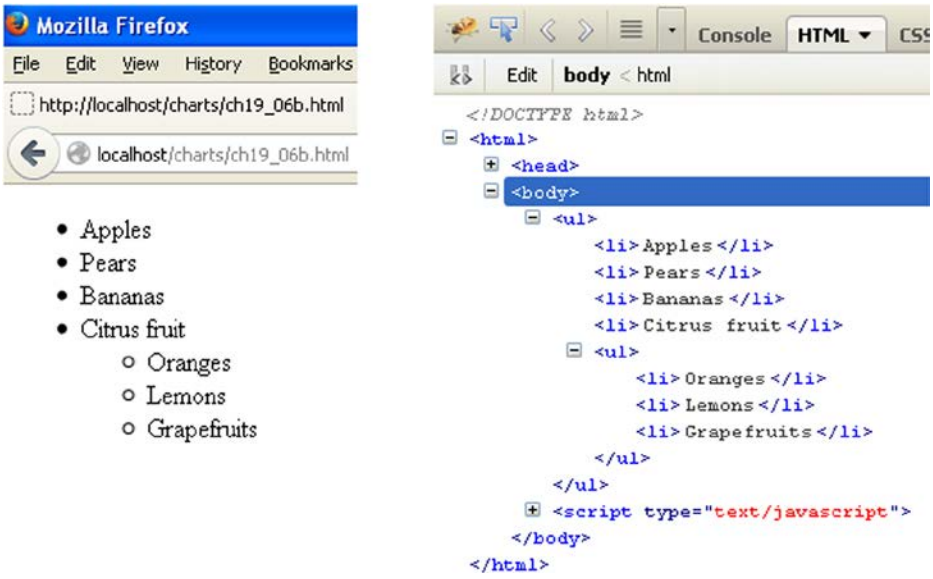


Figure 19-12. *FireBug shows the HTML structure with a nested unordered list in the browser on the left and in FireBug on the right*

The insert( ) Method

The last operator, `insert()`, has a particular behavior. If you use it with only one argument, it behaves as if you were using `append()`. Normally, it is used with two arguments. The first indicates the tag to add, and the second is the matching tag where you want to insert it. In fact, replace `append()` with `insert()` in the preceding fruit list example, and you will obtain a different result, shown in Figure 19-13 (the original list is on the left and new one with Oranges added is on the right).

```
d3.select('ul').insert('li','li').text("Oranges");
```

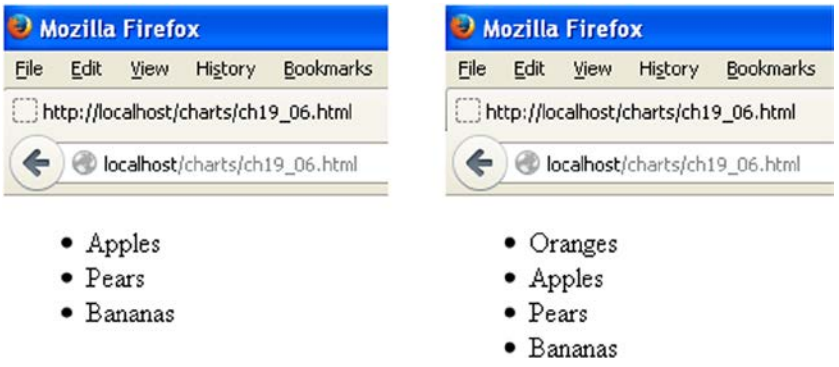


Figure 19-13. *Using the `insert()` operator, you can insert the Oranges item at the top of the list*

Now the new element is at the top of the unordered list. But if you wanted to insert a new item in a different location than the first? You can use the CSS selector `nth-child(i)` to do this, where `i` is the index of the element. Therefore, if you use the selector `li:nth-child(i)`, you are going to select the i-th `<li>` element. Thus, if you want to insert an element between the second and the third element, you need to call the third `<li>` element in the `insert()` operator (remember that this operator puts the new element before the one called):

```
d3.select('ul').insert('li','li:nth-child(2)').text("Oranges");
```

This will insert the new Orange item between the second and the third items in the list, as shown in Figure 19-14 (in the browser on the left and in FireBug on the right).

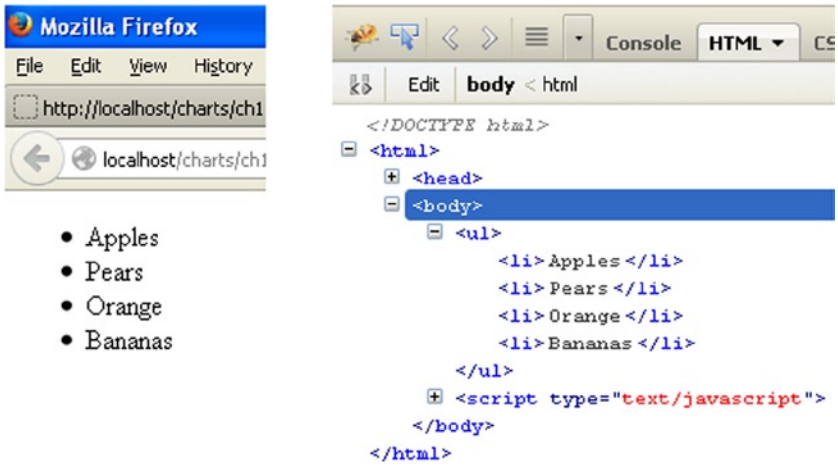


Figure 19-14. *Using the CSS selector nth-child, you can add the Oranges item in any position in the list*

HTML(), APPEND(), AND INSERT() OPERATORS: A BETTER UNDERSTANDING

Sometimes, understanding the functionality of these three operators isn't easy. Consider this schematic HTML structure, containing a generic parent tag and some children tags inside:

```
<parent>
    <child></child>
    <child></child>
    <child></child>
</parent>
```

The following simple diagrams show what each operator does exactly, in order to better understand the different behaviors. It is crucial that you fully understand the functionality of these three operators if you want to exploit the full potential of the D3 library.

When you need to create a new tag element at the end of a list of other tags at the same level of the HTML structure, use the `append()` operator. Figure 19-15 shows the behavior of this operator.

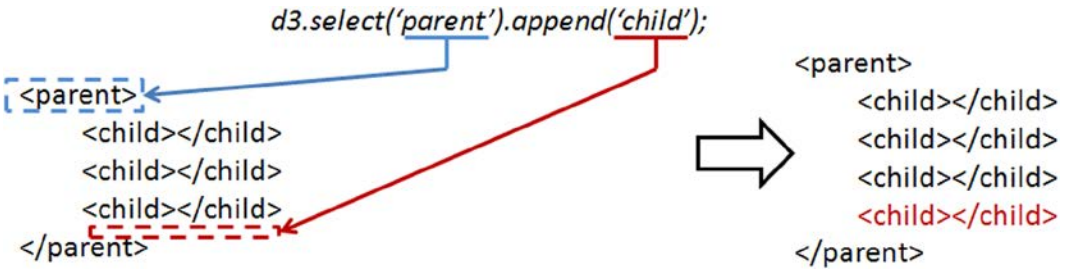


Figure 19-15. *The `append()` operator adds a child tag to the end of the list*

When you need to create a new tag element at the beginning of a list of other tags at the same level of the HTML structure, use the `insert()` operator. Figure 19-16 shows the behavior of this operator.

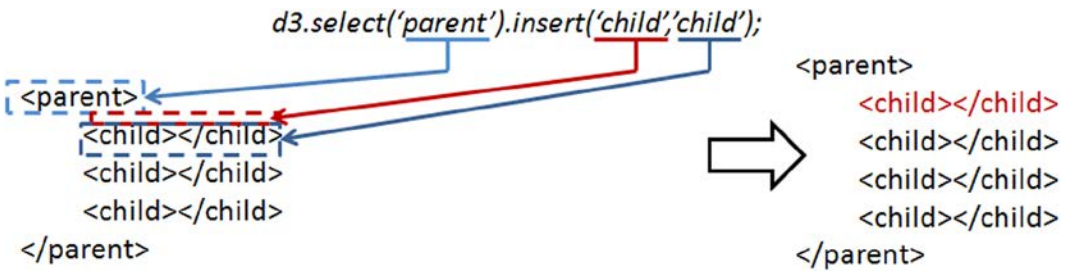


Figure 19-16. *The `insert()` operator adds a child tag before the child tag is passed as a second argument*

When you need to create a new tag element at a specific position in a list of other tags, always at the same level of the HTML structure, use the `insert()` operator. Figure 19-17 shows the behavior of this operator.

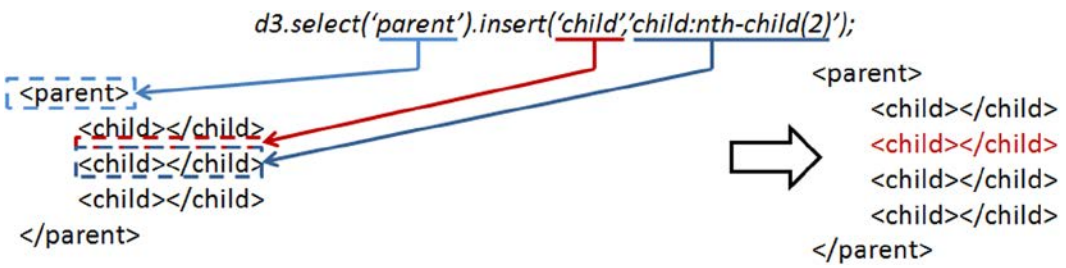


Figure 19-17. *You can pass a child of the list as the argument using the CSS selector nth-child()*

When you need to create a new tag element in place of another tag or in place of a list of other tags at the same level of the HTML structure, use the `html()` operator. Figure 19-18 illustrates the behavior of this operator.

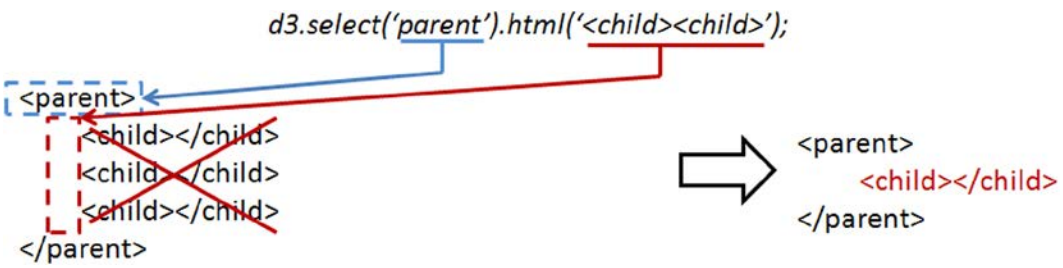


Figure 19-18. *The `html()` operator replaces the contents of the parent tag with the tag passed as the argument*

Inserting Data into Elements

You have just seen how to create new elements in your document. But how can you put the data inside? This is where the `data()` operator comes in, by passing an array of data as an argument.

For each element in the selection, a value will be assigned in the array following the same order of the sequence. This correspondence is indicated by a generic function in which `d` and `i` are passed as arguments.

```
function(d,i) {
    // code with d and i
    // return some elaboration of d;
}
```

This function will be executed as many times as there are elements in the list: `i` is the index of the sequence and `d` is the value in the data array corresponding to that index. Many times you are not interested in the value of `i` and use only `d`.

For those familiar with the `for` loop, it is as if you had written:

```
for(i=0; i < selection.length; i++){
    d = input_array[i];
    //  code with d and i
    //return output_array[i];
}
```

To understand the whole thing, there is no better way than to provide an example. Define an array containing the names of three fruits. You'll create an unordered list with three empty items and create a selection of these items with `selectAll()`. You must have a corresponding number of items in the selection and values in the array; otherwise, the values in surplus will not be evaluated. You associate the array to the selection and then, applying `function(d)`, write the values of each item within the list (see Listing 19-10).

Listing 19-10. ch19_07.html

```
<body>
<ul>
    <li></li>
    <li></li>
    <li></li>
</ul>
```

```
<script type="text/javascript">
    var fruits = ['Apples', 'Pears', 'Bananas'];

    d3.selectAll('li').data(fruits).text( function(d){
       return d;
    });
</script>
</body>
```

Figure 19-19 shows the result in the browser on the left and in FireBug on the right. In FireBug, you can see the HTML structure used for each list item `<li>` content, which was not present when you wrote Listing 19-10. These added text items are the values of the `fruits` array.

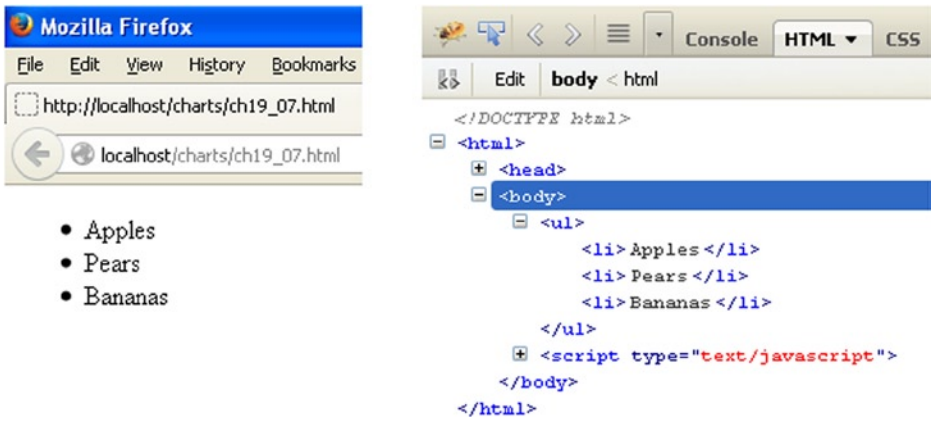


Figure 19-19. *It is possible to fill the content of HTML elements with array values*

The `data()` operator does not just bind data to elements, it computes a connection between the element of the selection and the data provided. All goes well as long as the length of the array is equal to the number of elements in the selection. But what if it is not so? If you have a selection with three elements and provide an array with five values, the two extra values will be stored in a special selection called "enter." This selection is accessible via the `enter()` operator on the return value of a call to data. You can see this in the example in Listing 19-11.

Listing 19-11. ch19_08.html

```
<body>
<ul>
    <li></li>
    <li></li>
    <li></li>
</ul>
<script type="text/javascript">
    var fruits = ['Apples', 'Pears', 'Bananas', 'Oranges', 'Strawberries'];
    var list = d3.select('ul');
    var fruits = list.selectAll('li').data(fruits);
    fruits.enter().append('li').text( function(d){
       return d;
    });
```

```
    fruits.text( function(d){
       return d;
    });
</script>
</body>
```

First, you define the array with five different fruits. Then, you make a selection that contains the list and assign it to the variable list. From this selection, you make a further selection containing the three empty list items and assign the `fruits` array to it. From this association, the last two values of the array will advance (Oranges and Strawberries), and thus they will be stored in the `enter` selection. Now you must pay particular attention to this point: usually it is best to deal with the `enter` selection first. Therefore, you have to access the `enter` selection and use `append()` in order to create two new list items with the two fruits advanced. Then you write the values in the fruit selection within the three existing list items.

You get a list with all five fruits, in the order in which they were entered. Figure 19-20 shows the change in the browser on the top and in FireBug on the bottom.

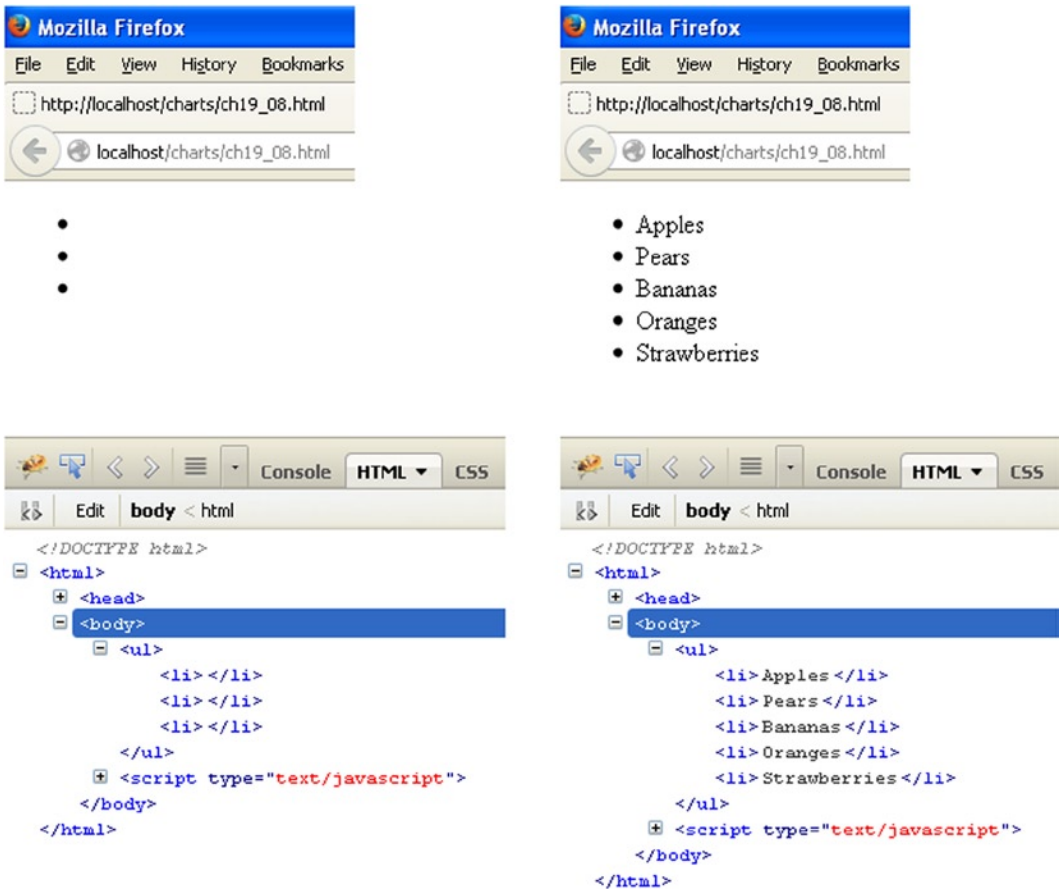


Figure 19-20. *It is possible to fill the content of HTML elements with array values and to integrate them with other elements if they are not enough*

Applying Dynamic Properties

You have seen how to define and modify styles, attributes, and other properties with the use of functions provided by the D3 framework. But so far, they have been treated as constants. It is time to take a leap forward. One of the advantages of the JavaScript language and especially of the D3 (and jQuery) library lies in its ability to make the content of a page dynamic. In fact, you have just seen how to delete, create, and manipulate the element tags in a web page. A similar approach is also applicable to other types of values such as CSS styles or the attributes of elements you created or manipulated through the selections mechanism. You could even create different options relating to events or controls.

D3 provides you with a set of specific functions for this purpose. Despite their apparent simplicity, these functions can be a powerful tool for those who know how to make full use of their mechanisms.

In the example in Listing 19-12, you use a generic function to assign a random color to the paragraphs. Every time the page is loaded, it shows a different set of colors.

Listing 19-12. ch19_09.html

```
<body>
<p>the first paragraph</p>
<p>the second paragraph</p>
<p>the third paragraph</p>
<p>the last paragraph</p>
<script>
d3.selectAll("p").style("color", function() {

    r = Math.round((Math.random() * 255));
    g = Math.round((Math.random() * 255));
    b = Math.round((Math.random() * 255));
    return "rgb("+r+", "+g+", "+b+")";

});
</script>
</body>
```

Figure 19-21 on the left shows the results of one loaded page and another, on the right, with different colors applied to it. Every time you load the page, you get a different color combination.

Figure 19-21. *The colors change each time the page loads*

Certainly, this is a very trivial example, but it shows the basic idea. Any value that you assign to an attribute, a text, or a style can be dynamically generated from a function.

Adding SVG Elements

You have finally arrived at the point where you can apply what you learned to create beautiful displays. In this section, you'll begin to learn about the peculiarities of the D3 library, with the creation and manipulation of graphic elements such as lines, squares, circles, and more. All of this will be done primarily by using nested structures of two tags: `<svg>` for graphic elements and `<g>` for application groups.

First, you'll learn how to create an SVG element and how to nest it in a group using the `<g>` tag. Later, you'll discover what SVG transformations are and how to apply them to groups of elements. Finally, with a further example, you'll see how to animate these elements with SVG transitions, in order to get nice animations.

Creating an SVG Element

You can start from a `<div>` tag, which will be used as a container for the visualization, similarly to what jQuery does with `<canvas>`. From this `<div>` tag, you create the root tag `<svg>` using the `append()` operator. Then you can set the size of the visualization by acting on the `height` and `width` attributes using the `attr()` operator (see Listing 19-13).

Listing 19-13. ch19_10.html

```
<body>
<div id="circle"></div>
<script type="text/javascript">
    var svg = d3.select('#circle')
       .append('svg')
       .attr('width', 200)
       .attr('height', 200);
</script>
</body>
```

From FireBug, you can see the `<body>` structure with the new `<svg>` element and its attributes (see Figure 19-22).

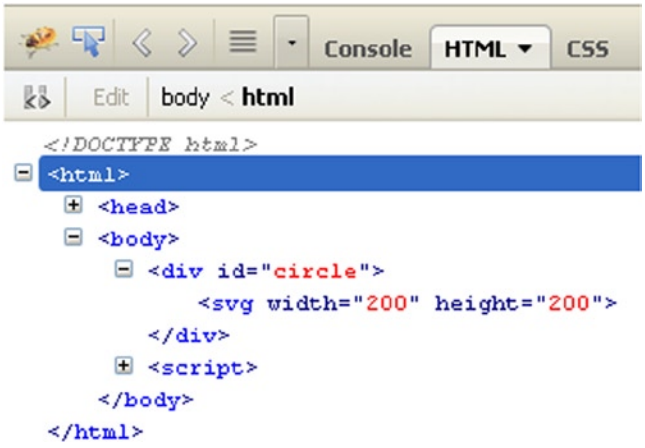


Figure 19-22. *FireBug shows the* `<svg>` *tag you just created*

You can also add a basic shape to the root tag <svg>. Let's add a yellow circle (see Listing 19-14). Once you understand this principle, it is very simple to repeat it whenever you wish.

Listing 19-14. ch19_10.html

```
<script type="text/javascript">
    var svg = d3.select('#circle')
       .append('svg')
       .attr('width', 200)
       .attr('height', 200);

    svg.append('circle')
       .style('stroke', 'black')
       .style('fill', 'yellow')
       .attr('r', 40)
       .attr('cx', 50)
       .attr('cy', 50);
</script>
```

Figure 19-23 shows the perfect yellow circle.

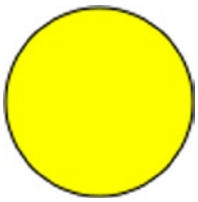


Figure 19-23. *A perfect yellow circle*

In FireBug, you can see how the tree structure of the tags gradually takes shape from the root <svg>, specifying all the attributes (see Figure 19-24).

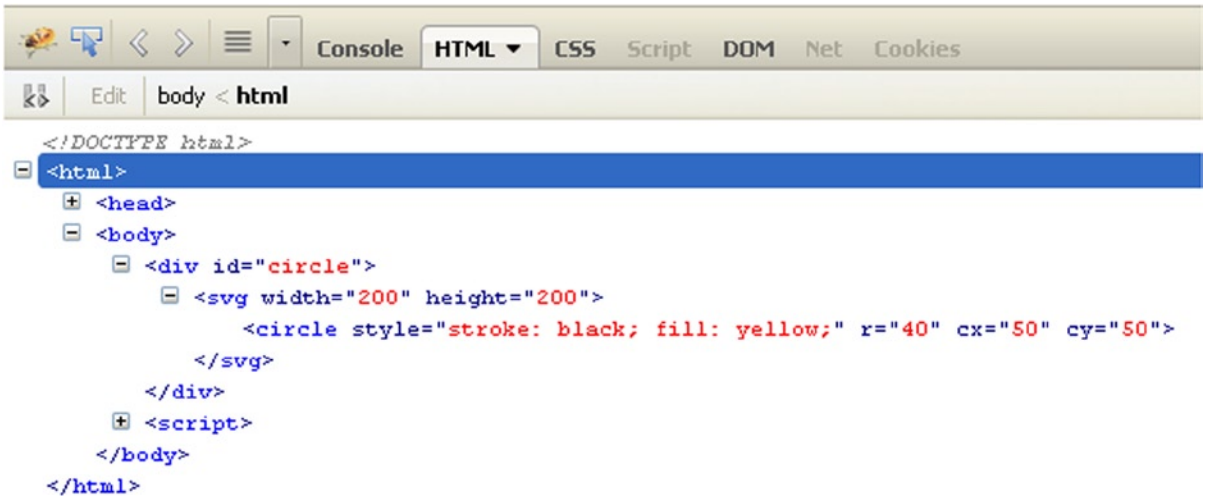


Figure 19-24. *In FireBug, it is possible to follow the development of the tag structure*

Now that you have seen how to create graphics using SVG tags, the next step is to apply transformations to them.

Transformations

A key aspect of D3 is its transformation capability. This extends the concept of SVG transformations in JavaScript. Once an object is created in SVG, from a simple square to more complex structures, it can be subjected to various transformations. The most common transformations include:

- **Scale**
- **Translate**
- **Rotate**

■ **Note** If you are interested in learning more about transformations, I suggest that you visit this page: https://developer.mozilla.org/en-US/docs/Web/SVG/Attribute/transform. It lists all the available transformations, with simple explanations.

Typically, you use sequences of these basic transformations to obtain more complex transformations. As always, you'll see a series of small examples to illustrate the concept of transformations. First, you'll draw a small red square in the same way you drew the yellow circle (see Listing 19-15). For this purpose, you use the `<rect>` tag. The only difference from `<circle>` is that for rectangles, you need to specify the position of the rectangle's top-left corner with x and y instead of the center of the circle. Then you have to specify the size of the rectangle, and since it is a square, the sides will be equal.

Listing 19-15. ch19_11a.html

```
<div id="square"></div>
<script type="text/javascript">
    var svg = d3.select('#square')
       .append('svg')
       .attr('width', 200)
       .attr('height', 200);
    svg.append('rect')
       .style('stroke', 'black')
       .style('fill', 'red')
       .attr('x', 50)
       .attr('y', 50)
       .attr('width', 50)
       .attr('height', 50);
</script>
```

It's a good time to introduce another concept that will be useful when in dealing with SVG elements: *groups* of elements. You'll often need to apply a series of operations, including only the transformations at times, to a group of shapes or to a complex shape (consisting of multiple basic shapes). This is possible by grouping several items together in a group, which is reflected in SVG by putting all the elements in a tag `<g>`. So if you want to apply a transformation to the red square for example, you need to insert it within a group (see Listing 19-16).

Listing 19-16. ch19_11a.html

```
var svg = d3.select('#square')
    .append('svg')
    .attr('width', 200)
    .attr('height', 200);

var g = svg.append("svg:g");

g.append('rect')
    .style('stroke', 'black')
    .style('fill', 'red')
    .attr('x', 50)
    .attr('y', 50)
    .attr('width', 50)
    .attr('height', 50);
```

Figure 19-25 shows how the SVG structure appears in FireBug.

Figure 19-25. *FireBug shows the SVG structure corresponding to the red square*

In the browser, you'll a small red square like the one shown in Figure 19-26.

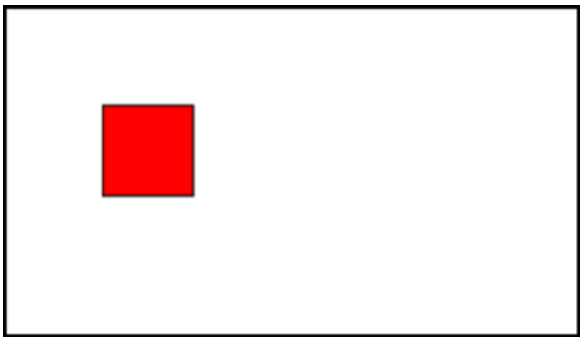


Figure 19-26. *A red square is a good object upon which to apply transformations*

Now you'll apply all the transformations, one by one. Start with the *translation*, which in SVG is expressed by the `translate(x, y)` function, where `x` and `y` are the amount of pixels by which the square will be moved (see Listing 19-17).

Listing 19-17. ch19_11b.html

```
var g = svg.append("svg:g")
    .attr("transform", "translate(" + 100 + ",0)");
```

Here I put the value `100` outside of the string passed as an attribute, to understand that at that point you can insert a previously defined variable. This will make the transformation more dynamic. With this line, you moved the square to the right 100 pixels (see Figure 19-27).

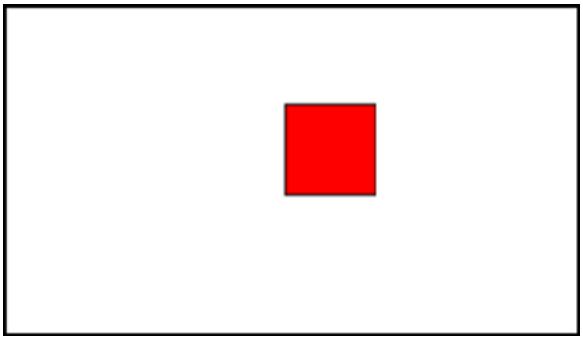


Figure 19-27. *Now the red square appears right-shifted by 100 pixels*

Another transformation that you can apply to the square is called *scaling*. In this SVG, it is expressed through the function `scale(s)` or `scale(sx, sy)`. If you pass a single parameter in the function, the scaling will be uniform, but if you pass two parameters, you can apply the expansion of the square in a different way horizontally and vertically. Listing 19-18 increases the size of the red square by two times. Thus, you need to apply the `scale()` transformation and to pass the value `2` as a parameter. The number passed is the factor by which the size of the square will be multiplied. Since you've passed a single parameter, scaling is uniform.

Listing 19-18. ch19_11c.html

```
var g = svg.append("svg:g")
    .attr("transform","scale(2)");
```

Figure 19-28 shows the square scaled by two times. The square has doubled in height and width.

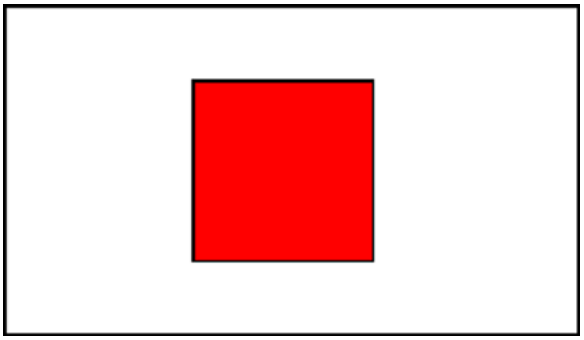


Figure 19-28. *The red square has doubled its size*

If you want non-uniform scaling, you can use something like Listing 19-19 to obtain a result similar to Figure 19-29. Non-uniform scaling can distort a figure to give another picture. In this case, you get a rectangle from a square.

Listing 19-19. ch19_11d.html

```
var g = svg.append("svg:g")
    .attr("transform","scale(2, 1)");
```

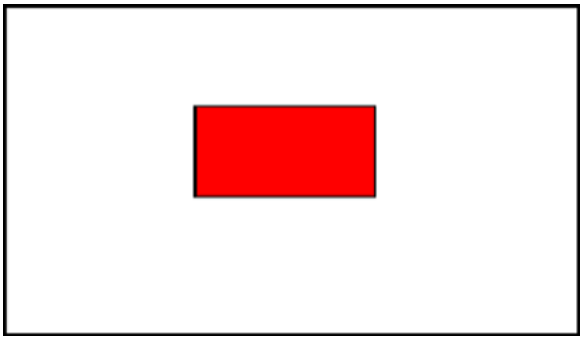


Figure 19-29. *A rectangle obtained by applying non-uniform scaling to a square*

The last kind of transformation is *rotation*. It is expressed in SVG with the function `rotate(degree,x,y)`, where the first argument is the angle of rotation (clockwise) in degrees, and `x` and `y` are the coordinates of the center of rotation.

Say you want the center of rotation to correspond with the center of the square, which is located at x = 75 and y = 75. If you wish to draw a rhombus, you need to perform a rotation of 45 degrees on the square (see Listing 19-20).

Listing 19-20. ch19_11e.html

```
var g = svg.append("svg:g")
    .attr("transform","rotate(45, 75, 75)");
```

You get the rhombus (see Figure 19-30).

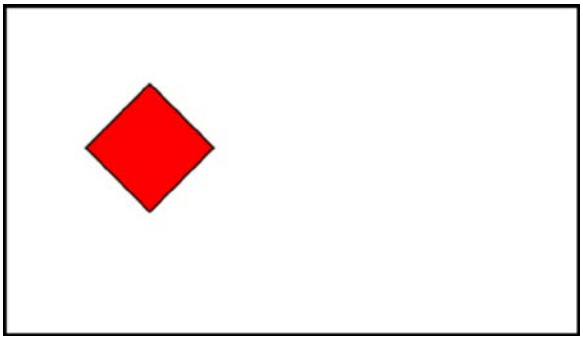


Figure 19-30. *A rhombus is the result you obtain when you rotate a square*

But the most interesting effect involves applying the transformations in a sequence, thereby creating a chain (see Listing 19-21).

Listing 19-21. ch19_11f.html

```
var g = svg.append("svg:g")
    .attr("transform", "translate(-30, 0),scale(2, 1),rotate(45, 75, 75)");
```

From this listing, you obtain the shape in Figure 19-31.

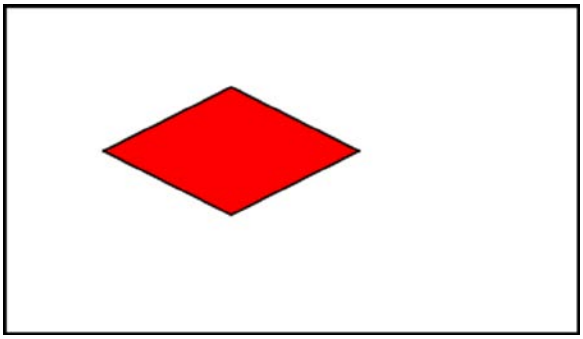


Figure 19-31. *A rhombus obtained by applying a chain of transformations to a square*

Transitions

You have seen that values of attributes, styles, and so forth, can be dynamic, according to the definition set with the help of certain functions. But D3 offers more—you can even animate your shapes. D3 provides three functions to this purpose:

- `transition()`
- `delay()`
- `duration()`

Naturally, you'll apply these functions to the SVG elements, thanks to D3, which can recognize any kind of values and interpolate them.

You define a transition when an SVG shape passes from one state to another. Both the starting state and the final state are characterized by several parameters that define the color, the shape, the size, and the position of an object. You take as the initial state the one defined in the yellow circle example (refer to Listing 19-14). In Listing 19-22, you subject the circle to a transition consisting of three different mutations: the circle changes its color to black (setting `fill` to `black`), it reduces its area (changing `r` from `40` to `10`), and it moves slightly to the right (changing `cx` from `50` to `150`).

Listing 19-22. ch19_12.html

```
<div id="circle"></div>
<script>
var svg = d3.select('#circle')
    .append('svg')
    .attr('width', 200)
    .attr('height', 200);

svg.append('circle')
    .style('stroke', 'black')
    .style('fill', 'yellow')
    .attr('r', 40)
    .attr('cx', 50)
    .attr('cy', 50)
    .transition()
    .delay(100)
    .duration(4000)
    .attr("r", 10)
    .attr("cx", 150)
    .style("fill", "black");
</script>
```

So, in this example, you add the `transition()` method to the methods chain. This separates the initial state from the final one and warns D3 of a transition. Immediately after the `transition()`, there are two other functions: `delay()` and `duration()`.

The `delay()` function takes one argument: the time that must elapse before the transition begins. The `duration()` function, in contrast, is defined as the time taken by the transition. The greater the value of the parameter passed, the slower the transition will be.

Following these three functions, you add all the attributes characterizing the final state of the figure to the method chain. D3 interpolates the intermediate values depending on the time you have established, and will generate all the intermediate figures with those values. What appears before your eyes is an animation in which the yellow circle turns black, moving to the left and decreasing in size. All of this takes four seconds.

Figure 19-32 shows the transition sequence whereby you can see the changes to the circle.

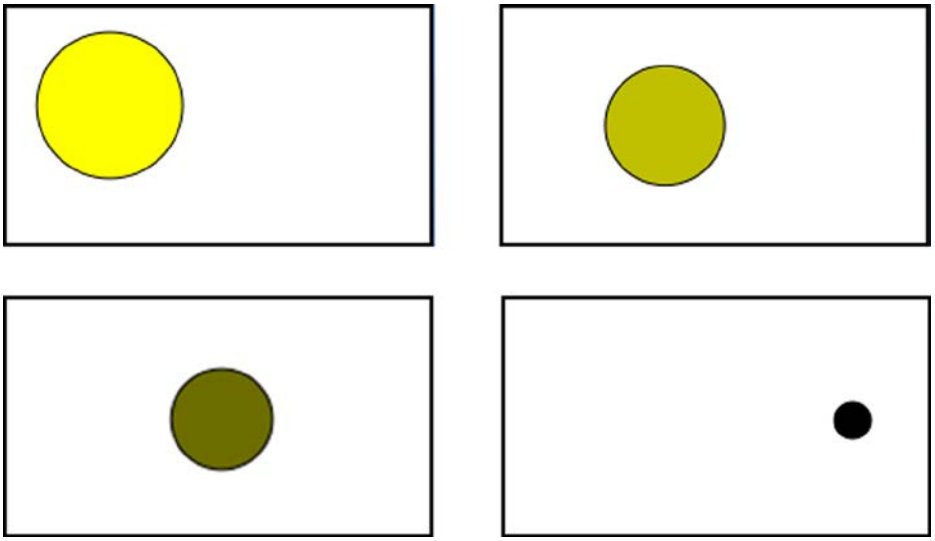


Figure 19-32. *Different instances of the animation of a circle subjected to transitions*

The simple examples you have seen so far were applied to one graphic element at a time. The next step is to apply what you have learned to groups of elements, so as to create more complex graphics. Subsequent chapters provide good examples in which this basic concept of the D3 library will be put into practice.

Summary

This chapter covered the highlights of the D3 library. Even without making use of the jQuery library, D3 can manage **selections**, **selectors** and **operators** in a very similar way. Through a series of examples, you have seen how to manipulate DOM elements by changing their attributes, and by creating new ones when needed. In the second part of the chapter, you learned what the main object of manipulations with the D3 library is: **SVG elements**. These are the graphic building blocks with which you build your charts. Finally, you took a quick look at how to apply **SVG transformations** to these graphic elements and then at how to exploit **SVG transitions** to generate nice animations.

The next chapter puts what you have learned so far about the D3 library into practice, by implementing line charts. Taking one SVG element after another, you'll see how to achieve similar results to those obtained with the jqPlot and Highcharts libraries.

CHAPTER 20

Line Charts with D3

In this chapter, you are going to create a line chart with ticks and labels. D3 is not a charting framework like jqPlot. It does allow you, however, to add Scalable Vector Graphics (SVG) elements to a document, and by manipulating these elements, you can create any kind of visualization. Such flexibility enables you to build any type of chart, building it up brick by brick.

You'll begin by looking at how to build the basic elements of a line chart using the D3 commands introduced in the previous chapter. In particular, you'll be analyzing the concepts of scales, domains, and ranges, which you'll be encountering frequently. These constitute a typical aspect of the D3 library, in terms of how it manages sets of values.

Once you understand how to manage values in their domains, scales, and intervals, you'll be ready to move on to the realization of the chart components, such as axes, axis labels, the title, and the grid. These components form the basis upon which you'll be drawing the line chart. Unlike in jqPlot, these components are not readily available but must be developed gradually. This will result in additional work, but it will also enable you to create special features. Your D3 charts will be able to respond to particular needs, or at least, they will have a totally original look. By way of example, you'll see how to add arrows to the axes.

Another peculiarity of the D3 library is the use of functions that read data contained in a file. You'll see how these functions work and how to exploit them for your needs.

Once you have the basic knowledge necessary to implement a line chart, you'll see how to implement a multiseries line chart. You'll also read about how to implement a legend and how to associate it with the chart.

Finally, to conclude, you'll analyze a particular case of line chart: the difference line chart. This will help you understand clip area paths—what they are and what their uses are.

Developing a Line Chart with D3

You'll begin to finally implement your chart using the D3 library. In this and in the following sections, you'll discover a different approach toward chart implementation compared to the one adopted with libraries such as jqPlot and Highcharts. Here the implementation is at a lower level and the code is much longer and more complex; however, nothing beyond your reach.

Now, step by step, or better, brick by brick, you'll discover how to produce a line chart and the elements that compose it.

Starting with the First Bricks

The first "brick" to start is to include the D3 library in your web page (for further information, see Appendix A):

```
<script src="../src/d3.v3.min.js"></script>
```

Or if you prefer to use a content delivery network (CDN) service:

```
<script src="http://d3js.org/d3.v3.min.js"></script>
```

The next "brick" consists of the input data array in Listing 20-1. This array contains the y values of the data series.

Listing 20-1. ch20_01.html

```
var data = [100, 110, 140, 130, 80, 75, 120, 130, 100];
```

Listing 20-2 you define a set of variables related to the size of the visualization where you're drawing the chart. The `w` and `h` variables are the width and height of the chart; the margins are used to create room at the edges of the chart.

Listing 20-2. ch20_01.html

```
w = 400;
h = 300;
margin_x = 32;
margin_y = 20;
```

Because you're working on a graph based on an x-axis and y-axis, in D3 it is necessary to define a scale, a domain, and a range of values for each of these two axes. Let's first clarify these concepts and learn how they are managed in D3.

Scales, Domains, and Ranges

You have already had to deal with scales, even though you might not realize it. The linear scale is more natural to understand, although in some examples, you have used a logarithmic scale (see the sidebar, "The Log Scale" in Chapter 9). A **scale** is simply a function that converts a value in a certain interval, called a **domain**, into another value belonging to another interval, called a **range**. But what does all this mean exactly? How does this help you?

Actually, this can serve you every time you want to affect the conversion of a value between two variables belonging to different intervals, but keeping its "meaning" with respect to the current interval. This relates to the concept of normalization.

Suppose that you want to convert a value from an instrument, such as the voltage reported by a multimeter. You know that the voltage value would read between 0 and 5 volts, which is the range of values, also known as the domain.

You want to convert the voltage measured on a scale of red. Using Red-Green-Blue (RGB) codes, this value will be between 0 and 255. You have now defined another color range, which is the range.

Now suppose the voltage on the multimeter reads 2.7 volts, and the color scale shown in red corresponds to 138 (actually 137.7). You have just applied a linear scale for the conversion of values. Figure 20-1 shows the conversion of the voltage value into the corresponding R value on the RGB scale. This conversion operates within a linear scale, since the values are converted linearly.

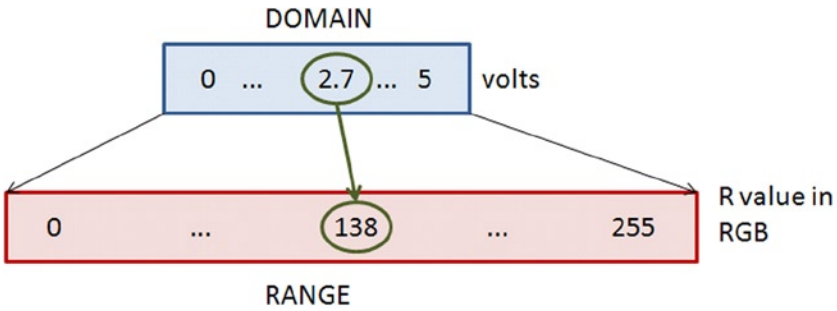


Figure 20-1. *The conversion from the voltage to the R value is managed by the D3 library*

But of what use is all of this? First, conversions between different intervals are not so uncommon when you aim to visualize data in a chart, and second, such conversions are managed completely by the D3 library. You do not need to do any calculations; you just need to define the domains, the range, and the scale to apply.

Translating this example into D3 code, you can write:

```
var scale = d3.scale.linear(),
            .domain([0,5]),
            .range([0,255]);
console.log(Math.round(scale(2.7)));      //it returns 138 on FireBug console
```

Inside the Code

You can define the scale, the domain, and the range; therefore, you can continue to implement the line chart by adding Listing 20-3 to your code.

Listing 20-3. ch20_01.html

```
y = d3.scale.linear().domain([0, d3.max(data)]).range([0 + margin_y, h - margin_y]);
x = d3.scale.linear().domain([0, data.length]).range([0 + margin_x, w - margin_x]);
```

Because the input data array is one-dimensional and contains the values that you want to represent on the y-axis, you can extend the domain from 0 to the maximum value of the array. You don't need to use a `for` loop to find this value. D3 provides a specific function called `max(date)`, where the argument passed is the array in which you want to find the maximum.

Now is the time to begin adding SVG elements. The first element to add is the `<svg>` element that represents the root of all the other elements you're going to add. The function of the `<svg>` tag is somewhat similar to that of the canvas in jQuery and jqPlot. As such, you need to specify the canvas size with `w` and `h`. Inside the `<svg>` element, you append a `<g>` element so that all the elements added to it internally will be grouped together.

Subsequently, apply a transformation to this group `<g>` of elements. In this case, the transformation consists of a translation of the coordinate grid, moving it down by `h` pixels, as shown in Listing 20-4.

Listing 20-4. ch20_01.html

```
var svg = d3.select("body")
    .append("svg:svg")
    .attr("width", w)
    .attr("height", h);

var g = svg.append("svg:g")
    .attr("transform", "translate(0," + h + ")");
```

Another fundamental thing you need in order to create a line chart is the **path element**. This path is filled with the data using the `d` attribute.

The manual entry of all these values is too arduous and in this regard D3 provides you with a function that does it for you: `d3.svg.line`. So, in Listing 20-5, you declare a variable called `line` in which every data is converted into a point (x, y).

Listing 20-5. ch20_01.html

```
var line = d3.svg.line()
    .x(function(d,i) { return x(i); })
    .y(function(d) { return -1 * y(d); });
```

As you'll see in all the cases where you need to make a scan of an array (a `for` loop), in D3 such a scan is handled differently through the use of the parameters `d` and `i`. The index of the current item of the array is indicated with `i`, whereas the current item is indicated with `d`. Recall that you translated the y-axis down with a transformation. You need to keep that mind; if you want to draw a line correctly, you must use the negative values of y. This is why you multiply the `d` values by -1.

The next step is to assign a line to a `path` element (see Listing 20-6).

Listing 20-6. ch20_01.html

```
g.append("svg:path").attr("d", line(data));
```

If you stopped here and launch the web browser on the page, you would get the image shown in Figure 20-2.

Figure 20-2. *The default behavior of an SVG path element is to draw filled areas*

This seems to be wrong somehow, but you must consider that in the creation of images with SVG, the role managed by CSS styles is preponderant. In fact, you can simply add the CSS classes in Listing 20-7 to have the line of data.

Listing 20-7. ch20_01.html

```
<style>
path {
    stroke: steelblue;
    stroke-width: 3;
    fill: none;
}

line {
    stroke: black;
}
</style>
```

Thus, with the CSS style classes suitably defined, you'll get a line as shown in Figure 20-3.

Figure 20-3. *The SVG path element draws a line if the CSS style classes are suitably defined*

But you are still far from having a line chart. You must add the two axes. To draw these two objects, you use simple SVG lines, as shown in Listing 20-8.

Listing 20-8. ch20_01.html

```
// draw the x axis
g.append("svg:line")
    .attr("x1", x(0))
    .attr("y1", -y(0))
    .attr("x2", x(w))
    .attr("y2", -y(0))

// draw the y axis
g.append("svg:line")
    .attr("x1", x(0))
    .attr("y1", -y(0))
    .attr("x2", x(0))
    .attr("y2", -y(d3.max(data))-10)
```

Now is the time to add labels. For this purpose there is a D3 function that greatly simplifies the job: `ticks()`. This function is applied to a D3 scale such as x or y, and returns rounded numbers to use as ticks. You need to use the function `text(String)` to obtain the string value of the current `d` (see Listing 20-9).

Listing 20-9. ch20_01.html

```
//draw the xLabels
g.selectAll(".xLabel")
    .data(x.ticks(5))
    .enter().append("svg:text")
    .attr("class", "xLabel")
    .text(String)
    .attr("x", function(d) { return x(d) })
    .attr("y", 0)
    .attr("text-anchor", "middle");

// draw the yLabels
g.selectAll(".yLabel")
    .data(y.ticks(5))
    .enter().append("svg:text")
    .attr("class", "yLabel")
    .text(String)
    .attr("x", 25)
    .attr("y", function(d) { return -y(d) })
    .attr("text-anchor", "end");
```

To align the labels, you need to specify the attribute `text-anchor`. Its possible values are `middle`, `start`, and `end`, depending on whether you want the labels aligned in the center, to the left, or to the right, respectively.

Here, you use the D3 function `attr()` to specify the attribute, but it is possible to specify it in the CSS style as well, as shown in Listing 20-10.

Listing 20-10. ch20_01.html

```
.xLabel {
    text-anchor: middle;
}

.yLabel {
    text-anchor: end;
}
```

In fact, writing these lines is pretty much the same thing. Usually, however, you'll prefer to set these values in the CSS style when you plan to change them—they are understood as parameters. Instead, in this case, or if you want them to be a fixed property of an object, it is preferable to insert them using the `attr()` function.

Now you can add the ticks to the axes. This is obtained by drawing a short line for each tick. What you did for tick labels you now do for ticks, as shown in Listing 20-11.

Listing 20-11. ch20_01.html

```
//draw the x ticks
g.selectAll(".xTicks")
    .data(x.ticks(5))
    .enter().append("svg:line")
    .attr("class", "xTicks")
    .attr("x1", function(d) { return x(d); })
    .attr("y1", -y(0))
```

```
    .attr("x2", function(d) { return x(d); })
    .attr("y2", -y(0)-5)

// draw the y ticks
g.selectAll(".yTicks")
    .data(y.ticks(5))
    .enter().append("svg:line")
    .attr("class", "yTicks")
    .attr("y1", function(d) { return -y(d); })
    .attr("x1", x(0)+5)
    .attr("y2", function(d) { return -y(d); })
    .attr("x2", x(0))
```

Figure 20-4 shows the line chart at this stage.

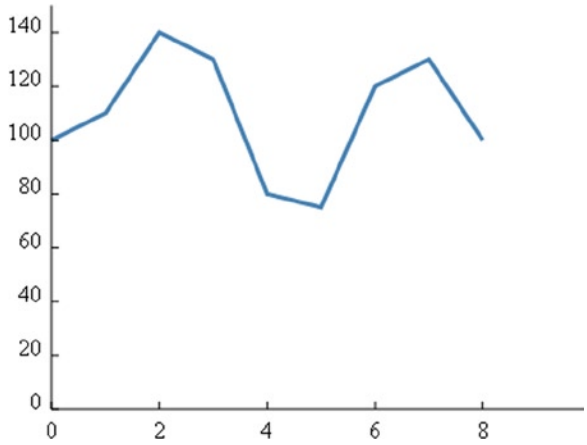


Figure 20-4. *Adding the two axes and the labels on them, you finally get a simple line chart*

As you can see, you already have a line chart. Perhaps by adding a grid, as shown in Listing 20-12, you can make things look better.

Listing 20-12. ch20_01.html

```
//draw the x grid
g.selectAll(".xGrids")
    .data(x.ticks(5))
    .enter().append("svg:line")
    .attr("class", "xGrids")
    .attr("x1", function(d) { return x(d); })
    .attr("y1", -y(0))
    .attr("x2", function(d) { return x(d); })
    .attr("y2", -y(d3.max(data))-10);
// draw the y grid
g.selectAll(".yGrids")
    .data(y.ticks(5))
    .enter().append("svg:line")
```

```
    .attr("class", "yGrids")
    .attr("y1", function(d) { return -y(d); })
    .attr("x1", x(w))
    .attr("y2", function(d) { return -y(d); })
    .attr("x2", x(0));
```

You can make a few small additions to the CSS style (see Listing 20-13) in order to get a light gray grid as the background of the line chart. Moreover, you can define the text style as it seems more appropriate, for example by selecting Verdana for the font, with size 9.

Listing 20-13. ch20_01.html

```
<style>
path {
    stroke: steelblue;
    stroke-width: 3;
    fill: none;
}
line {
    stroke: black;
}
.xGrids {
    stroke: lightgray;
}
.yGrids {
    stroke: lightgray;
}
text {
    font-family: Verdana;
    font-size: 9pt;
}
</style>
```

The line chart is now drawn with a light gray grid, as shown in Figure 20-5.

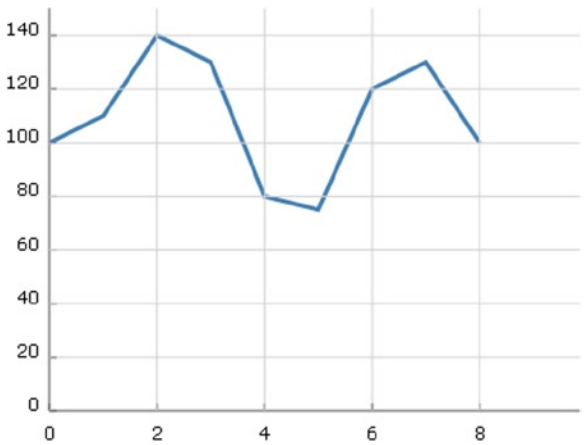


Figure 20-5. *A line chart with a grid covering the blue lines*

Look carefully at the Figure 20-5. The gray lines of the grid are drawn above the blue line representing the data. In other words, to be more explicit, you must be careful about the order in which you draw the SVG elements. In fact it is convenient to first draw the axes and the grid and then eventually to move on to the representation of the input data. Thus, you need to put all items that you want to draw in the right order, as shown in Listing 20-14.

Listing 20-14. ch20_01.html

```
<script>
var data = [100,110,140,130,80,75,120,130,100];
w = 400;
h = 300;
margin_x = 32;
margin_y = 20;
y = d3.scale.linear().domain([0, d3.max(data)]).range([0 + margin_y, h - margin_y]);
x = d3.scale.linear().domain([0, data.length]).range([0 + margin_x, w - margin_x]);
var svg = d3.select("body")
    .append("svg:svg")
    .attr("width", w)
    .attr("height", h);

var g = svg.append("svg:g")
    .attr("transform", "translate(0," + h + ")");

var line = d3.svg.line()
    .x(function(d,i) { return x(i); })
    .y(function(d) { return -y(d); });

// draw the y axis
g.append("svg:line")
    .attr("x1", x(0))
    .attr("y1", -y(0))
    .attr("x2", x(w))
    .attr("y2", -y(0));

// draw the x axis
g.append("svg:line")
    .attr("x1", x(0))
    .attr("y1", -y(0))
    .attr("x2", x(0))
    .attr("y2", -y(d3.max(data))-10);

//draw the xLabels
g.selectAll(".xLabel")
    .data(x.ticks(5))
    .enter().append("svg:text")
    .attr("class", "xLabel")
    .text(String)
    .attr("x", function(d) { return x(d) })
    .attr("y", 0)
    .attr("text-anchor", "middle");
```

```
// draw the yLabels
g.selectAll(".yLabel")
    .data(y.ticks(5))
    .enter().append("svg:text")
    .attr("class", "yLabel")
    .text(String)
    .attr("x", 25)
    .attr("y", function(d) { return -y(d) })
    .attr("text-anchor", "end");

//draw the x ticks
g.selectAll(".xTicks")
    .data(x.ticks(5))
    .enter().append("svg:line")
    .attr("class", "xTicks")
    .attr("x1", function(d) { return x(d); })
    .attr("y1", -y(0))
    .attr("x2", function(d) { return x(d); })
    .attr("y2", -y(0)-5);

// draw the y ticks
g.selectAll(".yTicks")
    .data(y.ticks(5))
    .enter().append("svg:line")
    .attr("class", "yTicks")
    .attr("y1", function(d) { return -1 * y(d); })
    .attr("x1", x(0)+5)
    .attr("y2", function(d) { return -1 * y(d); })
    .attr("x2", x(0));

//draw the x grid
g.selectAll(".xGrids")
    .data(x.ticks(5))
    .enter().append("svg:line")
    .attr("class", "xGrids")
    .attr("x1", function(d) { return x(d); })
    .attr("y1", -y(0))
    .attr("x2", function(d) { return x(d); })
    .attr("y2", -y(d3.max(data))-10);

// draw the y grid
g.selectAll(".yGrids")
    .data(y.ticks(5))
    .enter().append("svg:line")
    .attr("class", "yGrids")
    .attr("y1", function(d) { return -1 * y(d); })
    .attr("x1", x(w))
    .attr("y2", function(d) { return -y(d); })
    .attr("x2", x(0));
```

```
// draw the x axis
g.append("svg:line")
    .attr("x1", x(0))
    .attr("y1", -y(0))
    .attr("x2", x(w))
    .attr("y2", -y(0));

// draw the y axis
g.append("svg:line")
    .attr("x1", x(0))
    .attr("y1", -y(0))
    .attr("x2", x(0))
    .attr("y2", -y(d3.max(data))+10);

// draw the line of data points
g.append("svg:path").attr("d", line(data));
</script>
```

Figure 20-6 shows the line chart with the elements drawn in the correct sequence. In fact, the blue line representing the input data is now on the foreground covering the grid and not vice versa.

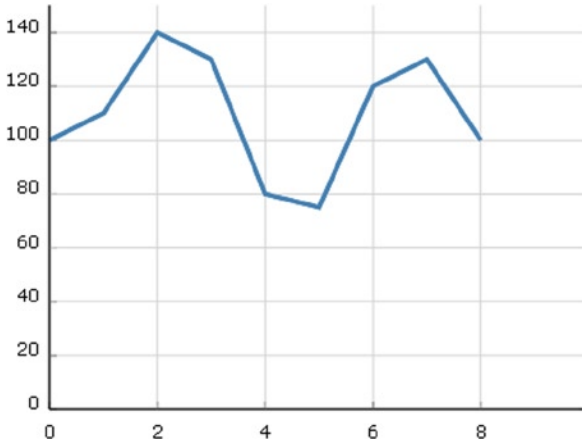


Figure 20-6. *A line chart with a grid drawn correctly*

Using Data with (x, y) Values

So far, you have used an input data array containing only the values of y. In general, you'll want to represent points that have x and y values assigned to them. Therefore, you'll extend the previous case by using the input data array in Listing 20-15.

Listing 20-15. ch20_02.html

```
var data = [{x: 0, y: 100}, {x: 10, y: 110}, {x: 20, y: 140},
            {x: 30, y: 130}, {x: 40, y: 80}, {x: 50, y: 75},
            {x: 60, y: 120}, {x: 70, y: 130}, {x: 80, y: 100}];
```

You can now see how the data is represented by dots containing both the values of x and y. When you use a sequence of data, you'll often need to immediately identify the maximum values of both x and y (and sometimes the minimum values too). In the previous case, you used the `d3.max` and `d3.min` functions, but these operate only on arrays, not on objects. The input data array you inserted is an array of objects. How do you solve this? There are several approaches. Perhaps the most direct way is to affect a scan of the data and find the maximum values for both x and y. In Listing 20-16, you define two variables that will contain the two maximums. Then scanning the values of x and y of each object at a time, you compare the current value of x and y with the values of `xMax` and `yMax`, in order to see which value is larger. The greater of the two will become the new maximum.

Listing 20-16. ch20_02.html

```
var xMax = 0, yMax = 0;
data.forEach(function(d) {
    if(d.x > xMax)
       xMax =  d.x;
    if(d.y > yMax)
       yMax =  d.y;
});
```

Several useful D3 functions work on arrays, so why not create two arrays directly from the input array of objects—one containing the values of x and the other containing the values of y? You can use these two arrays whenever necessary, instead of using the array of objects, which is far more complex (see Listing 20-17).

Listing 20-17. ch20_02.html

```
var ax = [];
var ay = [];
data.forEach(function(d,i){
     ax[i] = d.x;
     ay[i] = d.y;
})
var xMax = d3.max(ax);
var yMax = d3.max(ay);
```

This time you assign both x and y to the line of data points, as shown in Listing 20-18. This operation is very simple even when you're working with an array of objects.

Listing 20-18. ch20_02.html

```
var line = d3.svg.line()
    .x(function(d) { return x(d.x); })
    .y(function(d) { return -y(d.y); })
```

As for the rest of the code, there is not much to be changed—only a few corrections to the values of x and y bounds, as shown in Listing 20-19.

Listing 20-19. ch20_02.html

```
y = d3.scale.linear().domain([0, yMax]).range([0 + margin_y, h - margin_y]);
x = d3.scale.linear().domain([0, xMax]).range([0 + margin_x, w - margin_x]);
...
// draw the y axis
g.append("svg:line")
```

```
    .attr("x1", x(0))
    .attr("y1", -y(0))
    .attr("x2", x(0))
    .attr("y2", -y(yMax)-20)
...
//draw the x grid
g.selectAll(".xGrids")
    .data(x.ticks(5))
    .enter().append("svg:line")
    .attr("class", "xGrids")
    .attr("x1", function(d) { return x(d); })
    .attr("y1", -y(0))
    .attr("x2", function(d) { return x(d); })
    .attr("y2", -y(yMax)-10)

// draw the y grid
g.selectAll(".yGrids")
    .data(y.ticks(5))
    .enter().append("svg:line")
    .attr("class", "yGrids")
    .attr("y1", function(d) { return -1 * y(d); })
    .attr("x1", x(xMax)+20)
    .attr("y2", function(d) { return -1 * y(d); })
    .attr("x2", x(0))
```

Figure 20-7 shows the outcome resulting from the changes made to handle the y values introduced by the input data array.

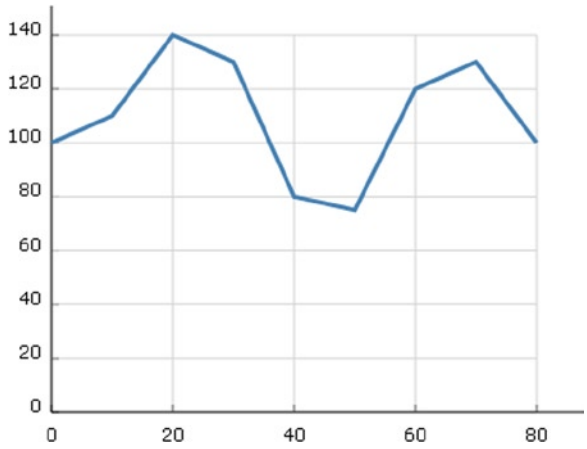


Figure 20-7. *A line chart with a grid and axis labels that take into account the y values entered with the input array*

Controlling the Axes' Range

In the line chart you just drew in the code, the data line will always be at the top of the chart. If your data oscillates at very high levels, with the scale of y starting at 0, you risk having a flattened trend line. It is also not optimal when the upper limit of the y-axis is the maximum value of y. Here, you'll add a check on the range of the axes. For this purpose in Listing 20-20, you define four variables that specify the lower and upper limits of the x- and y-axes.

Listing 20-20. ch20_03.html

```
var xLowLim = 0;
var xUpLim = d3.max(ax);
var yUpLim = 1.2 * d3.max(ay);
var yLowLim = 0.8 * d3.min(ay);
```

You consequently replace all the limit references with these variables. Note that the code becomes somewhat more readable. Specifying these four limits in a direct manner enables you to modify them easily as the need arises. In this case, only the range covered by the experimental data on y, plus a margin of 20%, is represented, as shown in Listing 20-21.

Listing 20-21. ch20_03.html

```
y = d3.scale.linear().domain([yLowLim, yUpLim]).range([0 + margin_y, h - margin_y]);
x = d3.scale.linear().domain([xLowLim, xUpLim]).range([0 + margin_x, w - margin_x]);

...

//draw the x ticks
g.selectAll(".xTicks")
    .data(x.ticks(5))
    .enter().append("svg:line")
    .attr("class", "xTicks")
    .attr("x1", function(d) { return x(d); })
    .attr("y1", -y(yLowLim))
    .attr("x2", function(d) { return x(d); })
    .attr("y2", -y(yLowLim)-5)

// draw the y ticks
g.selectAll(".yTicks")
    .data(y.ticks(5))
    .enter().append("svg:line")
    .attr("class", "yTicks")
    .attr("y1", function(d) { return -y(d); })
    .attr("x1", x(xLowLim))
    .attr("y2", function(d) { return -y(d); })
    .attr("x2", x(xLowLim)+5)

//draw the x grid
g.selectAll(".xGrids")
    .data(x.ticks(5))
    .enter().append("svg:line")
    .attr("class", "xGrids")
    .attr("x1", function(d) { return x(d); })
    .attr("y1", -y(yLowLim))
```

```
    .attr("x2", function(d) { return x(d); })
    .attr("y2", -y(yUpLim))

// draw the y grid
g.selectAll(".yGrids")
    .data(y.ticks(5))
    .enter().append("svg:line")
    .attr("class", "yGrids")
    .attr("y1", function(d) { return -y(d); })
    .attr("x1", x(xUpLim)+20)
    .attr("y2", function(d) { return -y(d); })
    .attr("x2", x(xLowLim))

// draw the x axis
g.append("svg:line")
    .attr("x1", x(xLowLim))
    .attr("y1", -y(yLowLim))
    .attr("x2", 1.2*x(xUpLim))
    .attr("y2", -y(yLowLim))

// draw the y axis
g.append("svg:line")
    .attr("x1", x(xLowLim))
    .attr("y1", -y(yLowLim))
    .attr("x2", x(xLowLim))
    .attr("y2", -1.2*y(yUpLim))
```

Figure 20-8 shows the new line chart with the y-axis range between 60 and 160, which displays the line better.

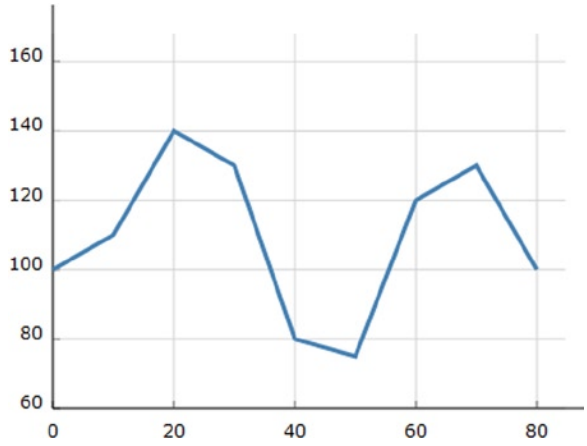


Figure 20-8. *A line chart with y-axis range focused around the y values*

Adding the Axis Arrows

In order to better understand the graphical versatility of D3, especially in the implementation of new features, you'll learn to add arrows to the x- and y-axes. To do this, you must add the two paths in Listing 20-22, as they will draw the arrows at the ends of both axes.

Listing 20-22. ch20_04.html

```
g.append("svg:path")
      .attr("class", "axisArrow")
      .attr("d", function() {
        var x1 = x(xUpLim)+23, x2 = x(xUpLim)+30;
        var y2 = -y(yLowLim),y1 = y2-3, y3 = y2+3
        return 'M'+x1+','+y1+','+x2+','+y2+','+x1+','+y3;
});

g.append("svg:path")
      .attr("class", "axisArrow")
      .attr("d", function() {
        var y1 = -y(yUpLim)-13, y2 = -y(yUpLim)-20;
        var x2 = x(xLowLim),x1 = x2-3, x3 = x2+3
        return 'M'+x1+','+y1+','+x2+','+y2+','+x3+','+y1;
});
```

In the CCS style, you add the `axisArrow` class, as shown in Listing 20-23. You can also choose to enable the `fill` attribute to obtain a filled arrow.

Listing 20-23. ch20_04.html

```
.axisArrow {
    stroke: black;
    stroke-width: 1;
    /*fill: black; */
}
```

Figure 20-9 shows the results, with and without filling.

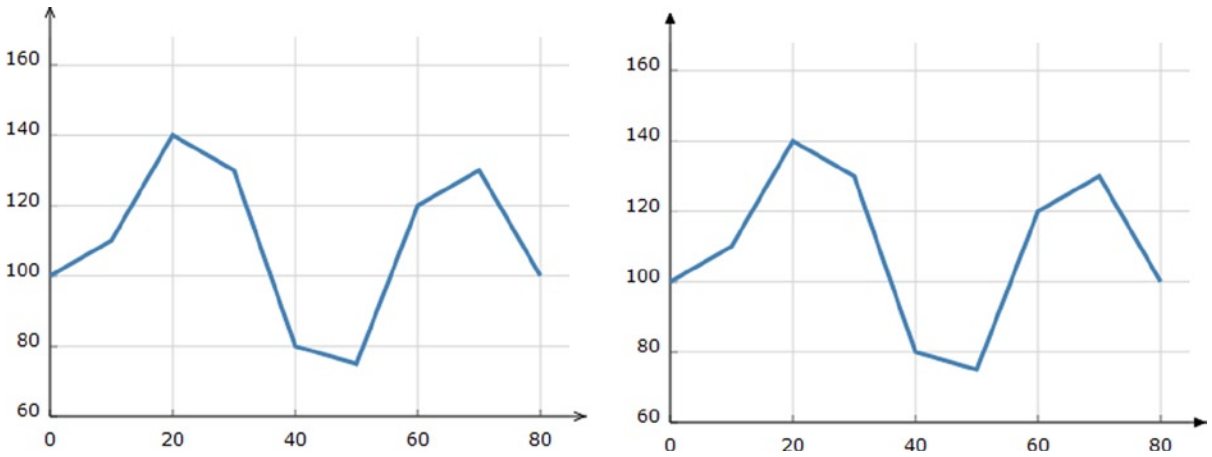


Figure 20-9. *Two different ways to represent the arrows on the axes*

Adding a Title and Axis Labels

In this section, you'll add a title to the chart. It is quite a simple thing to do, and you will use the SVG element called `text` with appropriate changes to the style, as shown in Listing 20-24. This code will place the title in the center, on top.

Listing 20-24. ch20_05.html

```
g.append("svg:text")
        .attr("x", (w / 2))
        .attr("y", -h + margin_y )
        .attr("text-anchor", "middle")
        .style("font-size", "22px")
        .text("My first D3 line chart");
```

Figure 20-10 shows the title added to the top of the line chart.

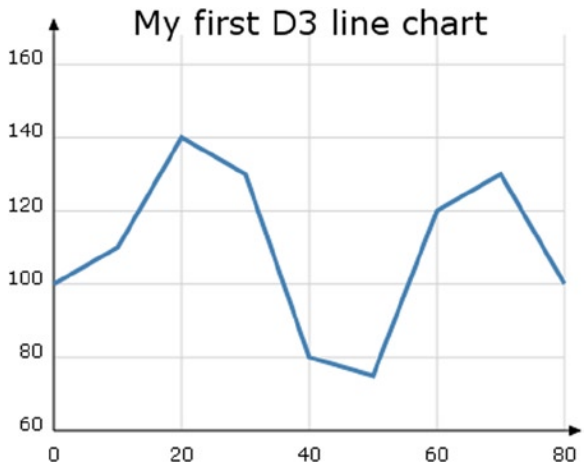


Figure 20-10. *A line chart with a title*

Following a similar procedure, you can add labels to the axes as well (see Listing 20-25).

Listing 20-25. ch20_05.html

```
g.append("svg:text")
        .attr("x", 25)
        .attr("y", -h + margin_y)
        .attr("text-anchor", "end")
        .style("font-size", "11px")
        .text("[#]");

g.append("svg:text")
        .attr("x", w - 40)
        .attr("y", -8 )
        .attr("text-anchor", "end")
        .style("font-size", "11px")
        .text("time [s]");
```

Figure 20-11 shows the two new axis labels put beside their corresponding axes.

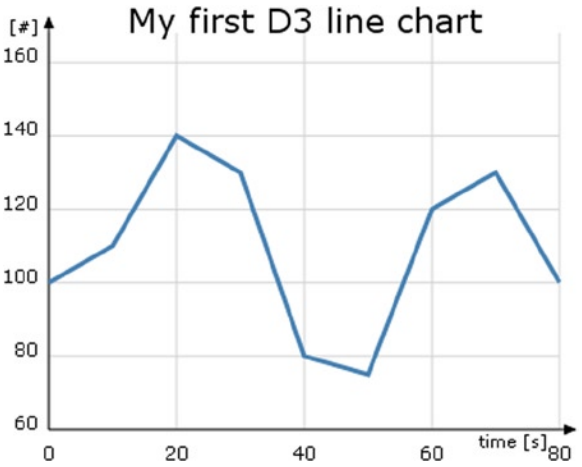


Figure 20-11. *A more complete line chart with title and axes labels*

Now that you have learned how to make a line chart, you're ready to try some more complex charts. Generally, the data you want to display in a chart are not present in the web page, but rather in external files. You'll integrate the following session on how to read data from external files with what you have learned so far.

Drawing a Line Chart from Data in a CSV File

When designing a chart, you typically refer to data of varied formats. This data often come from several different sources. In the most common case, you have applications on the server (which your web page is addressed to) that extract data from a database or by instrumentation, or you might even have data files collected in these servers. The example here uses a comma-separated value (CSV) file residing on the server as a source of data. This CSV file contains the data and could be loaded directly on the server or, as is more often the case, could be generated by other applications.

It is no coincidence that D3 has been prepared to deal with this type of file. For this purpose, D3 provides the function `d3.csv()`. You'll learn more about this topic with an example.

Reading and Parsing Data

First of all, you need to define the size of the "canvas," or better, the size and margins of the area where you want to draw the chart. This time, you define four margins. This will give you more control over the drawing area (see Listing 20-26).

Listing 20-26. ch20_06a.html

```
<!DOCTYPE html>
<meta charset="utf-8">
<style>
</style>
<body>
<script src="http://d3js.org/d3.v3.js"></script>
<script>
```

```
var margin = {top: 70, right: 20, bottom: 30, left: 50},
    w = 400 - margin.left - margin.right,
    h = 400 - margin.top - margin.bottom;
```

Now you deal with the data; write the data in Listing 20-27 with a text editor into a file and save it as `data_01.csv`.

Listing 20-27. data_01.csv

```
date,attendee
12-Feb-12,80
27-Feb-12,56
02-Mar-12,42
14-Mar-12,63
30-Mar-12,64
07-Apr-12,72
18-Apr-12,65
02-May-12,80
19-May-12,76
28-May-12,66
03-Jun-12,64
18-Jun-12,53
29-Jun-12,59
```

This data contain two sets of values separated by a comma (recall that CSV stands for comma-separated values). The first is in date format and lists the days on which there was a particular event, such as a conference or a meeting. The second column lists the number of attendees. Note that the dates are not enclosed in any quote marks.

In a manner similar to jqPlot, D3 has a number of tools that the control time formats. In fact, to handle dates contained in the CSV file, you must specify a parser, as shown in Listing 20-28.

Listing 20-28. ch20_06a.html

```
var parseDate = d3.time.format("%d-%b-%y").parse;
```

Here you need to specify the format contained in the CSV file: `%d` indicates the number format of the days, `%b` indicates the month reported with the first three characters, and `%y` indicates the year reported with the last two digits. You can specify the x and y values, assigning them with a scale and a range, as shown in Listing 20-29.

Listing 20-29. ch20_06a.html

```
var x = d3.time.scale().range([0, w]);
var y = d3.scale.linear().range([h, 0]);
```

Now that you have dealt with the correct processing of input data, you can begin to create the graphical components.

Implementing Axes and the Grid

You'll begin by learning how to graphically realize the two Cartesian axes. In this example, shown in Listing 20-30, you follow the most appropriate way to specify the x-axis and y-axis through the function `d3.svg.axis()`.

Listing 20-30. ch20_06a.html

```
var xAxis = d3.svg.axis()
    .scale(x)
    .orient("bottom")
    .ticks(5);

var yAxis = d3.svg.axis()
    .scale(y)
    .orient("left")
    .ticks(5);
```

This allows you to focus on the data, while all the axes-related concerns (ticks, labels, and so on) are automatically handled by the axis components. Thus, after you create `xAxis` and `yAxis`, you assign the scale of x and y to them and set the orientation. Is it simple? Yes; this time you don't have to specify all that tedious stuff about axes—their limits, where to put ticks and labels, and so on. Unlike the previous example, all this is automatically done with very few rows. I chose to introduce this concept now, because in the previous example, I wanted to emphasize the fact that every item you design is a brick that you can manage with D3, regardless of whether this process is then automated within the D3 library.

Now you can add the SVG elements to the page, as shown in Listing 20-31.

Listing 20-31. ch20_06a.html

```
var svg = d3.select("body").append("svg")
    .attr("width", w + margin.left + margin.right)
    .attr("height", h + margin.top + margin.bottom)
    .append("g")
    .attr("transform", "translate(" + margin.left + "," + margin.top + ")");

  svg.append("g")
      .attr("class", "x axis")
      .attr("transform", "translate(0," + h + ")")
      .call(xAxis);

  svg.append("g")
      .attr("class", "y axis")
      .call(yAxis);
```

Notice how the x-axis is subjected to translation. In fact, in the absence of specifications, the x-axis would be drawn at the top of the drawing area. Moreover, you also need to add to the CSS style. See Listing 20-32.

Listing 20-32. ch20_06a.html

```
<style>
body {
  font: 10px verdana;
}
.axis path,
.axis line {
  fill: none;
  stroke: #333;
}
</style>
```

Figure 20-12 shows the result.

Figure 20-12. *An empty chart ready to be filled with data*

Using FireBug, you can see the structure of the SVG elements as you have just defined them (see Figure 20-13).

```
<!DOCTYPE html>
<html>
  <head>
  <body>
    <script src="http://d3js.org/d3.v3.js">
    <script>
    <svg width="400" height="400">
      <g transform="translate(50,70)">
        <g class="x axis" transform="translate(0,300)">
        <g class="y axis">
      </g>
    </svg>
  </body>
</html>
```

Figure 20-13. *FireBug shows the structure of the SVG elements created dynamically to display the axes*

You can see that all the elements are automatically grouped within the group `<g>` tag. This gives you more control to apply possible transformations to the separate elements.

You can also add a grid if you want. You build the grid the same way you built the axes. In fact, in the same manner, you define two grid variables—`xGrid` and `yGrid`—using the `axis()` function in Listing 20-33.

Listing 20-33. ch20_06a.html

```
var yAxis = d3.svg.axis()
  ...
var xGrid = d3.svg.axis()
    .scale(x)
    .orient("bottom")
    .ticks(5)
    .tickSize(-h, 0, 0)
    .tickFormat("");

var yGrid = d3.svg.axis()
    .scale(y)
    .orient("left")
    .ticks(5)
    .tickSize(-w, 0, 0)
    .tickFormat("");
```

And in the bottom of the JavaScript code, you add the two new SVG elements to the others, as shown in Listing 20-34.

Listing 20-34. ch20_06a.html

```
svg.append("g")
    .attr("class", "y axis")
    .call(yAxis)

svg.append("g")
    .attr("class", "grid")
    .attr("transform", "translate(0," + h + ")")
    .call(xGrid);

svg.append("g")
    .attr("class", "grid")
    .call(yGrid);
```

Both elements are named with the same class name: `grid`. Thus, you can style them as a single element (see Listing 20-35).

Listing 20-35. ch20_06a.html

```
<style>
...
.grid .tick {
    stroke: lightgrey;
    opacity: 0.7;
}
.grid path {
    stroke-width: 0;
}
</style>
```

Figure 20-14 shows the horizontal grid lines you have just defined as SVG elements.

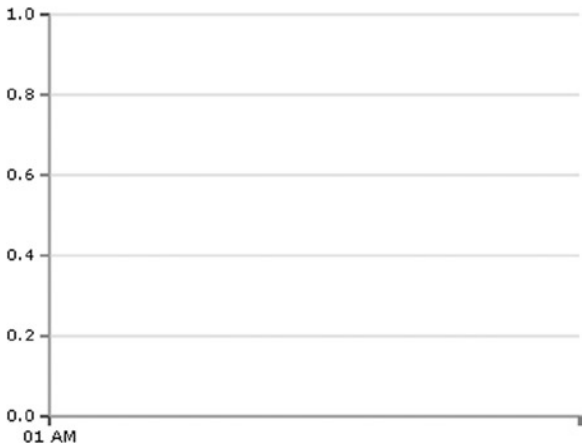


Figure 20-14. *Beginning to draw the horizontal grid lines*

Your chart is now ready to display the data from the CSV file.

Drawing Data with the csv() Function

It's now time to display the data in the chart, and you can do so with the D3 function `d3.csv()`, as shown in Listing 20-36.

Listing 20-36. ch20_06a.html

```
d3.csv("data_01.csv", function(error, data) {
// Here we will put all the SVG elements affected by the data
// on the file!!!
});
```

The first argument is the name of the CSV file; the second argument is a function where all the data within the file is handled. All the D3 functions that are in some way influenced by these values must be placed in this function. For example, you use `svg.append()` to create new SVG elements, but many of these functions need to know the x and y values of data. So you'll need to put them inside the `csv()` function as a second argument.

All the data in the CSV file is collected in an object called `data`. The different fields of the CSV file are recognized through their headers. The first thing you'll add is an iterative function where the `data` object is read item by item. Here, date values are parsed. You must ensure that all attendee values are read as numeric (this can be done by assigning each value to itself with a plus sign before it). See Listing 20-37.

Listing 20-37. ch20_06a.html

```
d3.csv("data_01.csv", function(error, data) {
    data.forEach(function(d) {
       d.date = parseDate(d.date);
       d.attendee = +d.attendee;
    });
});
```

Only now is it possible to define the domain on x and y in Listing 20-38, because only now do you know the values of this data.

Listing 20-38. ch20_06a.html

```
d3.csv("data_01.csv", function(error, data) {
    data.forEach(function(d) {
    ...
    });

    x.domain(d3.extent(data, function(d) { return d.date; }));
    y.domain(d3.extent(data, function(d) { return d.attendee; }));
});
```

Once data from the file is read and collected, it constitutes a set of points (x, y) that must be connected by a line. You'll use the SVG element `path` to build this line, as shown in Listing 20-39. As you saw previously, the function `d3.svg.line()` makes the work easier.

Listing 20-39. ch20_06a.html

```
d3.csv("data_01.csv", function(error, data) {
    data.forEach(function(d) {
    ...
    });
...
var line = d3.svg.line()
    .x(function(d) { return x(d.date); })
    .y(function(d) { return y(d.attendee); });
});
```

You can also add the two axes labels and a title to the chart. This is a good example of how to build `<g>` groups manually. Previously the group and all the elements inside were created by functions; now you need to do this explicitly. If you wanted to add the two axis labels to one group and the title to another, you'd need to specify two different variables: `labels` and `title` (see Listing 20-40).

Listing 20-40. ch20_06a.html

```
d3.csv("data_01.csv", function(error, data) {
    data.forEach(function(d) {
    ...
    });
...
    var labels = svg.append("g")
       .attr("class","labels")

    labels.append("text")
       .attr("transform", "translate(0," + h + ")")
       .attr("x", (w-margin.right))
       .attr("dx", "-1.0em")
       .attr("dy", "2.0em")
       .text("[Months]");
    labels.append("text")
       .attr("transform", "rotate(-90)")
       .attr("y", -40)
       .attr("dy", ".71em")
       .style("text-anchor", "end")
```

```
        .text("Attendees");

    var title = svg.append("g")
        .attr("class","title");

    title.append("text")
        .attr("x", (w / 2))
        .attr("y", -30 )
        .attr("text-anchor", "middle")
        .style("font-size", "22px")
        .text("A D3 line chart from CSV file");
});
```

In each case you create an SVG element <g> with the `append()` method, and define the group with a class name. Subsequently, you assign the SVG elements to the two groups, using `append()` on these variables.

Finally, you can add the `path` element, which draws the line representing the data values (see Listing 20-41).

Listing 20-41. ch20_06a.html

```
d3.csv("data_01.csv", function(error, data) {
    data.forEach(function(d) {
    ...
    });
...

    svg.append("path")
        .datum(data)
        .attr("class", "line")
        .attr("d", line);
});
```

Even for this new SVG element, you must not forget to add its CSS style settings, as shown in Listing 20-42.

Listing 20-42. ch20_06a.html

```
<style>
...
.line {
    fill: none;
    stroke: steelblue;
    stroke-width: 1.5px;
}
</style>
```

Figure 20-15 shows the nice line chart reporting all the data in the CSV file.

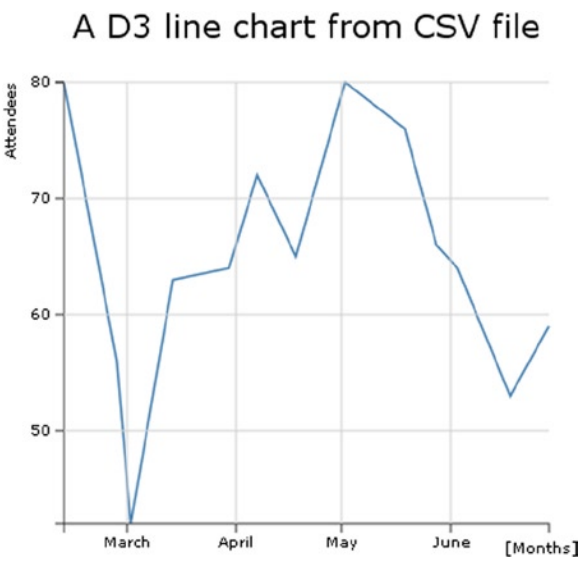


Figure 20-15. *A complete line chart with all of its main components*

Adding Marks to the Line

As you have seen for the line chart of jqPlot, even here it is possible to make further additions. For example, you could put data marker on the line.

Inside the `d3.csv()` function at the end of all the added SVG elements, you can add markers (see Listing 20-43). Remember that these elements depend on the data, so they must be inserted inside the `csv()` function.

Listing 20-43. ch20_06b.html

```
d3.csv("data_01.csv", function(error, data) {
    data.forEach(function(d) {
    ...
    });
...

    svg.selectAll(".dot")
       .data(data)
       .enter().append("circle")
       .attr("class", "dot")
       .attr("r", 3.5)
       .attr("cx", function(d) { return x(d.date); })
       .attr("cy", function(d) { return y(d.attendee); });
});
```

In the style part of the file, you add the CSS style definition of the `.dot` class in Listing 20-44.

Listing 20-44. ch20_06b.html

```
.dot {
    stroke: steelblue;
    fill: lightblue;
}
```

Figure 20-16 shows a line chart with small circles as markers; this result is very similar to the one obtained with the jqPlot library.

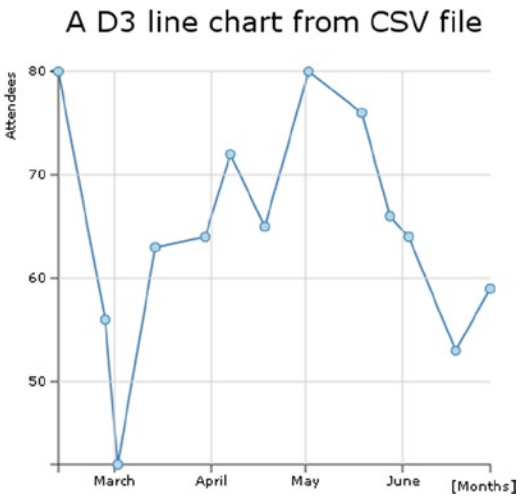


Figure 20-16. *A complete line chart with markers*

These markers have a circle shape, but it is possible to give them many other shapes and colors. For example, you can use markers with a square shape (see Listing 20-45).

Listing 20-45. ch20_06c.html

```
<style>
.dot {
    stroke: darkred;
    fill: red;
}
</style>
...
svg.selectAll(".dot")
    .data(data)
    .enter().append("rect")
    .attr("class", "dot")
    .attr("width", 7)
    .attr("height", 7)
    .attr("x", function(d) { return x(d.date)-3.5; })
    .attr("y", function(d) { return y(d.attendee)-3.5; });
```

Figure 20-17 shows the same line chart, but this time it uses small red squares for markers.

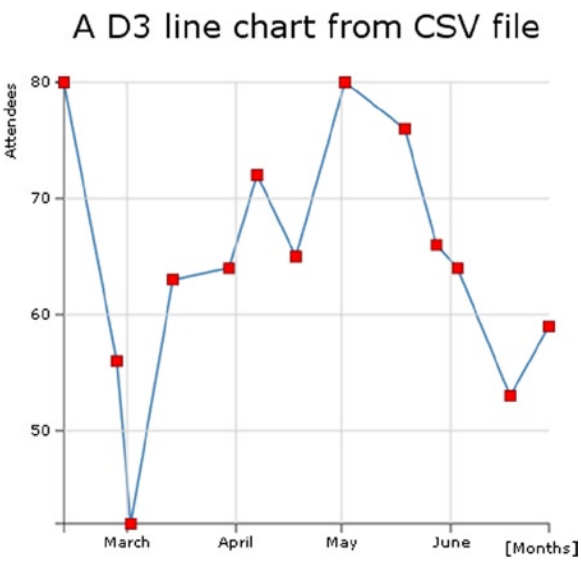


Figure 20-17. *One of the many marker options*

You could also use markers in the form of a yellow rhombus, often referred to as diamonds (see Listing 20-46).

Listing 20-46. ch20_06d.html

```
<style>
.dot {
    stroke: orange;
    fill: yellow;
}
</style>
...
svg.selectAll(".dot")
    .data(data)
    .enter().append("rect")
    .attr("class", "dot")
    .attr("transform", function(d) {
       var str = "rotate(45," + x(d.date) + "," + y(d.attendee) + ")";
       return str;
    })
    .attr("width", 7)
    .attr("height", 7)
    .attr("x", function(d) { return x(d.date)-3.5; })
    .attr("y", function(d) { return y(d.attendee)-3.5; });
```

Figure 20-18 shows markers in the form of a yellow rhombus.

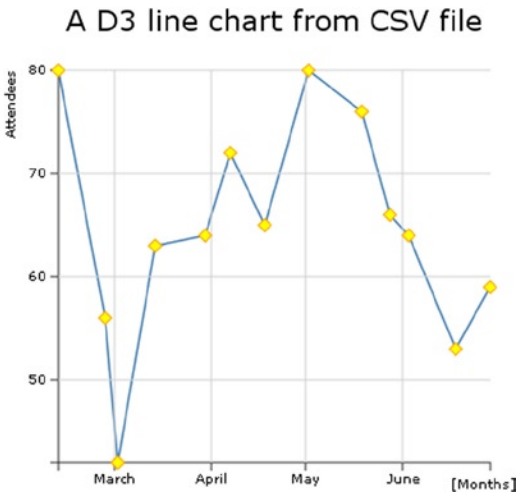


Figure 20-18. *Another marker option*

Line Charts with Filled Areas

In this section, you put point markers aside and return to the basic line chart. Another interesting feature you can add to your chart is to fill in the area below the line. Do you remember the `d3.svg.line()` function? Well, here you are using the `d3.svg.area()` function. Just as you have a `line` object in D3, you have an `area` object as well. Therefore, to define an `area` object, you can add the rows in bold in Listing 20-47 to the code, in the section just below the definition of the `line` object.

Listing 20-47. ch20_07.html

```
var line = d3.svg.line()
    .x(function(d) { return x(d.date); })
    .y(function(d) { return y(d.attendee); });

var area = d3.svg.area()
    .x(function(d) { return x(d.date); })
    .y0(h)
    .y1(function(d) { return y(d.attendee); });

var labels = svg.append("g")
...
```

As you can see, to define an area you need to specify the three functions that delimit the edges: `x`, `y0`, and `y1`. In this case, `y0` is constant, corresponding to the bottom of the drawing area (the x-axis). You now need to create the corresponding element in SVG, which is represented by a `path` element, as shown in Listing 20-48.

Listing 20-48. ch20_07.html

```
d3.csv("data_01.csv", function(error, data) {

  data.forEach(function(d) {
    d.date = parseDate(d.date);
```

```
    d.attendee = +d.attendee;
  });

...
svg.append("path")
    .datum(data)
    .attr("class", "line")
    .attr("d", line);

svg.append("path")
    .datum(data)
    .attr("class", "area")
    .attr("d", area);
});
```

As shown in Listing 20-49, you need to specify the color setting in the corresponding CSS style class.

Listing 20-49. ch20_07.html

```
.area {
    fill: lightblue;
}
```

Figure 20-19 shows an area chart derived from the line chart.

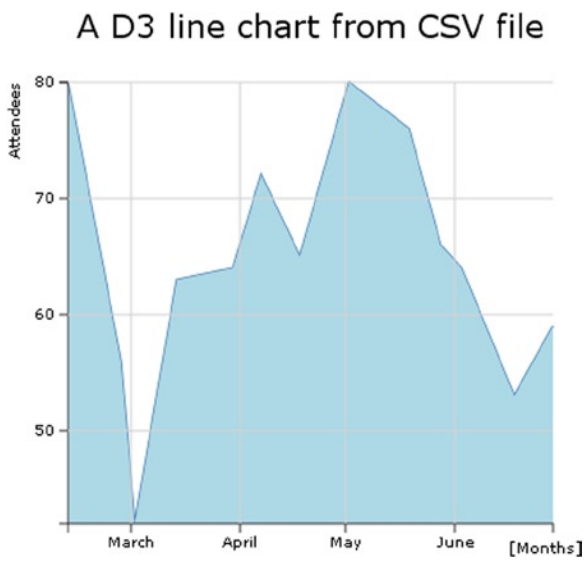


Figure 20-19. *An area chart*

Multiseries Line Charts

Now that you're familiar with the creation of the basic components of a line chart using SVG elements, the next step is to start dealing with multiple data series: multiseries line charts. The most important element covered in this section is the legend. You'll learn to create one by exploiting the basic graphical elements that SVG provides.

Working with Multiple Series of Data

So far, you've been working with a single series of data. It is now time to move on to multiseries. In the previous example, you used a CSV file as a source of data. Now, you'll look at another D3 function: `d3.tsv()`. It performs the same task as `csv()`, but operates on tab-separated value (TSV) files.

Copy Listing 20-50 into your text editor and save it as `data_02.tsv` (see the following note).

■ **Note** The values in a TSV file are tab-separated, so when you write or copy Listing 20-50, remember to check that there is only one tab character between each value.

Listing 20-50. data_02.tsv

```
Date    europa    asia    america
12-Feb-12    52    40    65
27-Feb-12    56    35    70
02-Mar-12    51    45    62
14-Mar-12    63    44    82
30-Mar-12    64    54    85
07-Apr-12    70    34    72
18-Apr-12    65    36    69
02-May-12    56    40    71
19-May-12    71    55    75
28-May-12    45    32    68
03-Jun-12    64    44    75
18-Jun-12    53    36    78
29-Jun-12    59    42    79
```

Listing 20-50 has four columns, where the first column is a date and the other three are values from different continents. The first column contains the x values; the others are the corresponding y values of the three series.

Start writing the code in Listing 20-51; there isn't any explanation, because this code is virtually identical to the previous example.

Listing 20-51. ch20_08a.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<script src="http://d3js.org/d3.v3.js"></script>
<style>
body {
    font: 10px verdana;
}
```

```
.axis path,
.axis line {
    fill: none;
    stroke: #333;
}

.grid .tick {
    stroke: lightgrey;
    opacity: 0.7;
}
.grid path {
    stroke-width: 0;
}

.line {
    fill: none;
    stroke: steelblue;
    stroke-width: 1.5px;
}
</style>
</head>
<body>
<script type="text/javascript">
var margin = {top: 70, right: 20, bottom: 30, left: 50},
    w = 400 - margin.left - margin.right,
    h = 400 - margin.top - margin.bottom;
var parseDate = d3.time.format("%d-%b-%y").parse;
var x = d3.time.scale().range([0, w]);
var y = d3.scale.linear().range([h, 0]);

var xAxis = d3.svg.axis()
    .scale(x)
    .orient("bottom")
    .ticks(5);

var yAxis = d3.svg.axis()
    .scale(y)
    .orient("left")
    .ticks(5);

var xGrid = d3.svg.axis()
    .scale(x)
    .orient("bottom")
    .ticks(5)
    .tickSize(-h, 0, 0)
    .tickFormat("");

var yGrid = d3.svg.axis()
    .scale(y)
    .orient("left")
    .ticks(5)
```

```
    .tickSize(-w, 0, 0)
    .tickFormat("");

var svg = d3.select("body").append("svg")
    .attr("width", w + margin.left + margin.right)
    .attr("height", h + margin.top + margin.bottom)
    .append("g")
    .attr("transform", "translate(" + margin.left + "," + margin.top + ")");

var line = d3.svg.line()
    .x(function(d) { return x(d.date); })
    .y(function(d) { return y(d.attendee); });

// Here we add the d3.tsv function
// start of the part of code to include in the d3.tsv() function
d3.tsv("data_02.tsv", function(error, data) {

    svg.append("g")
       .attr("class", "x axis")
       .attr("transform", "translate(0," + h + ")")
       .call(xAxis);

    svg.append("g")
       .attr("class", "y axis")
       .call(yAxis);

    svg.append("g")
       .attr("class", "grid")
       .attr("transform", "translate(0," + h + ")")
       .call(xGrid);

    svg.append("g")
       .attr("class", "grid")
       .call(yGrid);
});
//end of the part of code to include in the d3.tsv() function

var labels = svg.append("g")
    .attr("class","labels");

labels.append("text")
    .attr("transform", "translate(0," + h + ")")
    .attr("x", (w-margin.right))
    .attr("dx", "-1.0em")
    .attr("dy", "2.0em")
    .text("[Months]");
labels.append("text")
    .attr("transform", "rotate(-90)")
    .attr("y", -40)
    .attr("dy", ".71em")
    .style("text-anchor", "end")
    .text("Attendees");
```

```
var title = svg.append("g")
    .attr("class","title");

title.append("text")
    .attr("x", (w / 2))
    .attr("y", -30 )
    .attr("text-anchor", "middle")
    .style("font-size", "22px")
    .text("A multiseries line chart");
</script>
</body>
</html>
```

When you deal with multiseries data in a single chart, you need to be able to identify the data quickly and as a result you need to use different colors. D3 provides some functions generating an already defined sequence of colors. For example, there is the `category10()` function, which provides a sequence of 10 different colors. You can create a color set for the multiseries line chart just by writing the line in Listing 20-52.

Listing 20-52. ch20_08a.html

```
...
var x = d3.time.scale().range([0, w]);
var y = d3.scale.linear().range([h, 0]);
var color = d3.scale.category10();
var xAxis = d3.svg.axis()
...
```

You now need to read the data in the TSV file. As in the previous example, just after the call of the `d3.tsv()` function, you add a parser, as shown in Listing 20-53. Since you have to do with date values on the x-axis, you have to parse this type of value. You'll use the **`parseDate()`** function.

Listing 20-53. ch20_08a.html

```
d3.tsv("data_02.tsv", function(error, data) {
    data.forEach(function(d) {
       d.date = parseDate(d.date);
    });
...
});
```

You've defined a color set, using the `category10()` function in a chain with a `scale()` function. This means that D3 handles color sequence as a scale. You need to create a domain, as shown in Listing 20-54 (in this case it will be composed of discrete values, not continuous values such as those for x or y). This domain consists of the headers in the TSV file. In this example, you have three continents. Consequently, you'll have a domain of three values and a color sequence of three colors.

Listing 20-54. ch20_08a.html

```
d3.tsv("data_02.tsv", function(error, data) {
    data.forEach(function(d) {
       d.date = parseDate(d.date);
    });
```

```
    color.domain(d3.keys(data[0]).filter(function(key) {
       return key !== "date";
    }));
...
});
```

In Listing 20-53, you can see that `data[0]` is passed as an argument to the `d3.keys()` function. `data[0]` is the object corresponding to the first row of the TSV file:

```
Object { date=Date {Sun Feb 12 2012 00:00:00 GMT+0100},
europa="52", asia="40", america="65"}.
```

The `d3.keys()` function extracts the name of the values from inside an object, the same name which we find as a header in the TSV file. So using `d3.keys(data[0])`, you get the array of strings:

```
["date","europa","asia","america"]
```

You are interested only in the last three values, so you need to filter this array in order to exclude the key `"date"`. You can do so with the `filter()` function. Finally, you'll assign the three continents to the domain of colors.

```
["europa","asia","america"]
```

The command in Listing 20-55 reorganizes all the data in an array of structured objects. This is done by the function `map()` with an inner function, which maps the values following a defined structure.

Listing 20-55. ch20_08a.html

```
d3.tsv("data_02.tsv", function(error, data) {
...
    color.domain(d3.keys(data[0]).filter(function(key) {
       return key !== "date";
    }));

    var continents = color.domain().map(function(name) {
       return {
          name: name,
          values: data.map(function(d) {
             return {date: d.date, attendee: +d[name]};
          })
       };
    });
...
});
```

So this is the array of three objects called continents.

```
[ Object { name="europa", values=[13]},
  Object { name="asia", values=[13]},
  Object { name="america", values=[13]} ]
```

Every object has a continent name and a values array of 13 objects:

```
[ Object { date=Date, attendee=52 },
  Object { date=Date, attendee=56 },
  Object { date=Date, attendee=51 },
  ...]
```

You have the data structured in a way that allows for subsequent handling. In fact, when you need to specify the y domain of the chart, you can find the maximum and minimum of all values (not of each single one) in the series with a double iteration (see Listing 20-56). With `function(c)`, you make an iteration of all the continents and with `function(v)`, you make an iteration of all values inside them. In the end, `d3.min` and `d3.max` will extract only one value.

Listing 20-56. ch20_08a.html

```
d3.tsv("data_02.tsv", function(error, data) {
...
    var continents = color.domain().map(function(name) {
    ...
   });

    x.domain(d3.extent(data, function(d) { return d.date; }));
    y.domain([
       d3.min(continents, function(c) {
          return d3.min(c.values, function(v) { return v.attendee; });
       }),
       d3.max(continents, function(c) {
           return d3.max(c.values, function(v) { return v.attendee; });
       })
    ]);
...
});
```

Thanks to the new data structure, you can add an SVG element <g> for each continent containing a line path, as shown in Listing 20-57.

Listing 20-57. ch20_08a.html

```
d3.tsv("data_02.tsv", function(error, data) {
...

    svg.append("g")
       .attr("class", "grid")
       .call(yGrid);

    var continent = svg.selectAll(".continent")
       .data(continents)
       .enter().append("g")
       .attr("class", "continent");
```

```
    continent.append("path")
       .attr("class", "line")
       .attr("d", function(d) { return line(d.values); })
       .style("stroke", function(d) { return color(d.name); });
});
```

The resulting multiseries line chart is shown in Figure 20-20.

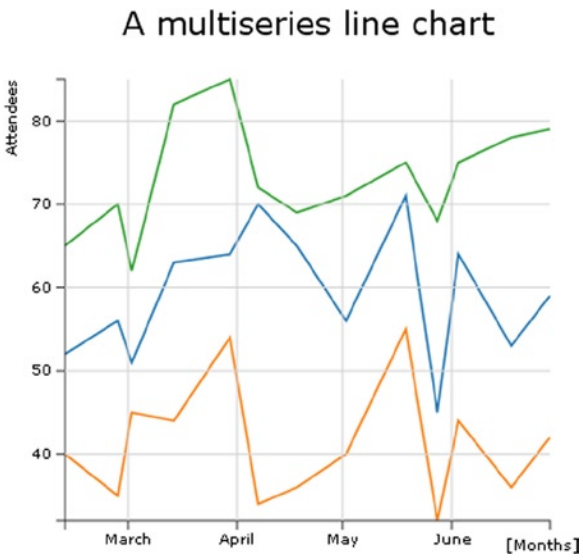


Figure 20-20. *A multiseries line chart*

Adding a Legend

When you are dealing with multiseries charts, the next logical step is to add a legend in order to categorize the series with colors and labels. Since a legend is a graphical object like any other, you need to add the SVG elements that allow you to draw it on the chart (see Listing 20-58).

Listing 20-58. ch20_08a.html

```
d3.tsv("data_02.tsv", function(error, data) {
...
    continent.append("path")
       .attr("class", "line")
       .attr("d", function(d) { return line(d.values); })
       .style("stroke", function(d) { return color(d.name); });

    var legend = svg.selectAll(".legend")
       .data(color.domain().slice().reverse())
       .enter().append("g")
       .attr("class", "legend")
       .attr("transform", function(d, i) { return "translate(0," + i * 20 + ")"; });
```

```
    legend.append("rect")
       .attr("x", w - 18)
       .attr("y", 4)
       .attr("width", 10)
       .attr("height", 10)
       .style("fill", color);

    legend.append("text")
       .attr("x", w - 24)
       .attr("y", 9)
       .attr("dy", ".35em")
       .style("text-anchor", "end")
       .text(function(d) { return d; });
});
```

The resulting multiseries line chart is shown in Figure 20-21, with a legend.

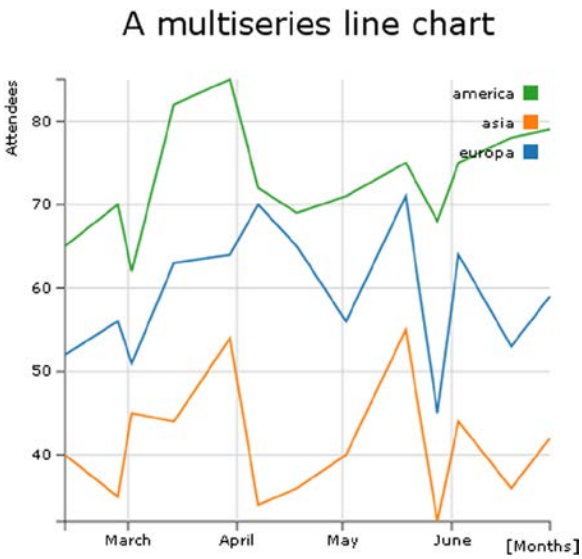


Figure 20-21. *A multiseries line chart with a legend*

Interpolating Lines

Do you remember the smooth effect on the lines when you tackled multiseries line charts with the jqPlot library? (If not, you can find it in the "Smooth Line Chart" section in Chapter 9.) In a line chart ,you usually have the data points connected by sequence, one by one, in a straight line. You have also seen how to join the points into a curved line. In fact, the effect was obtained via an interpolation. The D3 library covers interpolations of data points in a more correct way from a mathematical point of view. You, therefore, need to delve a little deeper into this concept.

When you have a set of values and want to represent them using a line chart, you essentially want to know the trend that these values suggest. From this trend, you can evaluate which values may be obtained at intermediate points between a data point and the next. Well, with such an estimate you are actually affecting an **interpolation**. Depending on the trend and the degree of accuracy you want to achieve, you can use various mathematical methods that regulate the shape of the curve that will connect the data points.

The most commonly used method is the spline. (If you want to deepen your knowledge of the topic, visit http://paulbourke.net/miscellaneous/interpolation/.) Table 20-1 lists the various types of interpolation that the D3 library makes available.

Table 20-1. *The options for interpolating lines available within the D3 library*

Options	Description
`basis`	A B-spline, with control point duplication on the ends.
`basis-open`	An open B-spline; may not intersect the start or end.
`basis-closed`	A closed B-spline, as in a loop.
`bundle`	Equivalent to `basis`, except the tension parameter is used to straighten the spline.
`cardinal`	A Cardinal spline, with control point duplication on the ends.
`cardinal-open`	An open Cardinal spline; may not intersect the start or end, but will intersect other control points.
`cardinal-closed`	A closed Cardinal spline, as in a loop.
`Linear`	Piecewise linear segments, as in a polyline.
`linear-closed`	Close the linear segments to form a polygon.
`monotone`	Cubic interpolation that preserves a monotone effect in y.
`step-before`	Alternate between vertical and horizontal segments, as in a step function.
`step-after`	Alternate between horizontal and vertical segments, as in a step function.

You find these options by visiting https://github.com/mbostock/d3/wiki/SVG-Shapes#wiki-line_interpolate.

Now that you understand better what an interpolation is, you can see a practical case. In the previous example, you had three series represented by differently colored lines and made up of segments connecting the data points (x,y). But it is possible to draw corresponding interpolating lines instead.

As shown in Listing 20-59, you just add the `interpolate()` method to the `d3.svg.line` to get the desired effect.

Listing 20-59. ch20_08b.html

```
var line = d3.svg.line()
    .interpolate("basis")
    .x(function(d) { return x(d.date); })
    .y(function(d) { return y(d.attendee); });
```

Figure 20-22 shows the interpolating lines applied to the three series in the chart. The straight lines connecting the data points have been replaced by curves.

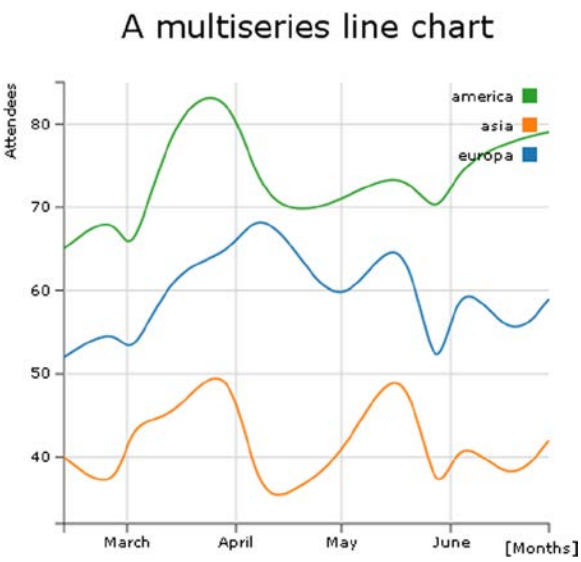


Figure 20-22. *A smooth multiseries line chart*

Difference Line Chart

This kind of chart portrays the area between two series. In the range where the first series is greater than the second series, the area has one color, where it is less the area has a different color. A good example of this kind of chart compares the trend of income and expense across time. When the income is greater than the expenses, the area will be green (usually the green color stands for OK), whereas when it is less, the area is red (meaning BAD). Write the values in Listing 20-60 in a TSV (or CSV) file and name it `data_03.tsv` (see the note).

■ **Note** The values in a TSV file are tab-separated, so when you write or copy Listing 20-60, remember to check that there is only one tab character between each value.

Listing 20-60. data_03.tsv

```
Date    income    expense
12-Feb-12    52    40
27-Feb-12    56    35
02-Mar-12    31    45
14-Mar-12    33    44
30-Mar-12    44    54
07-Apr-12    50    34
18-Apr-12    65    36
02-May-12    56    40
19-May-12    41    56
28-May-12    45    32
03-Jun-12    54    44
18-Jun-12    43    46
29-Jun-12    39    52
```

Start writing the code in Listing 20-61; explanations aren't included this time, as the example is virtually identical to the previous one.

Listing 20-61. ch20_09.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<script src="http://d3js.org/d3.v3.js"></script>
<style>
body {
  font: 10px verdana;
}
.axis path,
.axis line {
  fill: none;
  stroke: #333;
}
.grid .tick {
    stroke: lightgrey;
    opacity: 0.7;
}
.grid path {
    stroke-width: 0;
}
</style>
</head>
<body>
<script type="text/javascript">

var margin = {top: 70, right: 20, bottom: 30, left: 50},
    w = 400 - margin.left - margin.right,
    h = 400 - margin.top - margin.bottom;

var parseDate = d3.time.format("%d-%b-%y").parse;
var x = d3.time.scale().range([0, w]);
var y = d3.scale.linear().range([h, 0]);

var xAxis = d3.svg.axis()
    .scale(x)
    .orient("bottom")
    .ticks(5);

var yAxis = d3.svg.axis()
    .scale(y)
    .orient("left")
    .ticks(5);

var xGrid = d3.svg.axis()
    .scale(x)
    .orient("bottom")
```

```
    .ticks(5)
    .tickSize(-h, 0, 0)
    .tickFormat("");

var yGrid = d3.svg.axis()
    .scale(y)
    .orient("left")
    .ticks(5)
    .tickSize(-w, 0, 0)
    .tickFormat("");

var svg = d3.select("body").append("svg")
    .attr("width", w + margin.left + margin.right)
    .attr("height", h + margin.top + margin.bottom)
    .append("g")
    .attr("transform", "translate(" + margin.left + "," + margin.top + ")");

// Here we add the d3.tsv function
// start of the part of code to include in the d3.tsv() function
d3.tsv("data_03.tsv", function(error, data) {

    svg.append("g")
       .attr("class", "x axis")
       .attr("transform", "translate(0," + h + ")")
       .call(xAxis);

    svg.append("g")
       .attr("class", "y axis")
       .call(yAxis);

    svg.append("g")
       .attr("class", "grid")
       .attr("transform", "translate(0," + h + ")")
       .call(xGrid);

    svg.append("g")
       .attr("class", "grid")
       .call(yGrid);

});
//end of the part of code to include in the d3.tsv() function

var labels = svg.append("g")
    .attr("class","labels");

labels.append("text")
    .attr("transform", "translate(0," + h + ")")
    .attr("x", (w-margin.right))
    .attr("dx", "-1.0em")
    .attr("dy", "2.0em")
    .text("[Months]");
labels.append("text")
```

```
    .attr("transform", "rotate(-90)")
    .attr("y", -40)
    .attr("dy", ".71em")
    .style("text-anchor", "end")
    .text("Millions ($)");

var title = svg.append("g")
    .attr("class","title");

title.append("text")
    .attr("x", (w / 2))
    .attr("y", -30 )
    .attr("text-anchor", "middle")
    .style("font-size", "22px")
    .text("A difference chart");
</script>
</body>
</html>
```

First you read the TSV file to check whether the income and expense values are positive. Then you parse all the date values (see Listing 20-62).

Listing 20-62. ch20_09.html

```
d3.tsv("data_03.tsv", function(error, data) {

    data.forEach(function(d) {
       d.date = parseDate(d.date);
       d.income = +d.income;
       d.expense = +d.expense;
    });
    ...
});
```

Here, unlike the example shown earlier (the multiseries line chart), there is no need to restructure the data, so you can create a domain on x and y, as shown in Listing 20-63. The maximum and minimum are obtained by comparing income and expense values with `Math.max` and `Math.min` at every step, and then finding the values affecting the iteration at each step with `d3.min` and `d3.max`.

Listing 20-63. ch20_09.html

```
d3.tsv("data_03.tsv", function(error, data) {
    data.forEach(function(d) {
    ...
    });

    x.domain(d3.extent(data, function(d) { return d.date; }));
    y.domain([
       d3.min(data, function(d) {return Math.min(d.income, d.expense); }),
       d3.max(data, function(d) {return Math.max(d.income, d.expense); })
    ]);
    ...
});
```

Before adding SVG elements, you need to define some CSS classes. You'll use the color red when expenses are greater than income, and green otherwise. You need to define these colors, as shown in Listing 20-64.

Listing 20-64. ch20_09.html

```
<style>
...
.area.above {
    fill: darkred;
}

.area.below {
    fill: lightgreen;
}

.line {
    fill: none;
    stroke: #000;
    stroke-width: 1.5px;
}
</style>
```

Since you need to represent lines and areas, you define them by using the interpolation between data points (see Listing 20-65).

Listing 20-65. ch20_09.html

```
d3.tsv("data_03.tsv", function(error, data) {
    ...

    svg.append("g")
       .attr("class", "grid")
       .call(yGrid);

    var line = d3.svg.area()
       .interpolate("basis")
       .x(function(d) { return x(d.date); })
       .y(function(d) { return y(d["income"]); });

    var area = d3.svg.area()
       .interpolate("basis")
       .x(function(d) { return x(d.date); })
       .y1(function(d) { return y(d["income"]); });

});
```

As you can see, you are actually defining only the line of income points; there is no reference to expense values. But you are interested in the area between the two lines of income and expenditure, so when you define the `path` element, in order to draw this area, you can put the expense values as a border, with a generic function iterating the `d` values (see Listing 20-66).

Listing 20-66. ch20_09.html

```
d3.tsv("data_03.tsv", function(error, data) {
    ...

    var area = d3.svg.area()
       .interpolate("basis")
       .x(function(d) { return x(d.date); })
       .y1(function(d) { return y(d["income"]); });

    svg.datum(data);

    svg.append("path")
       .attr("class", "area below")
       .attr("d", area.y0(function(d) { return y(d.expense); }));

    svg.append("path")
       .attr("class", "line")
       .attr("d", line);

});
```

If you load the web page now, you should get the desired area (see Figure 20-23).

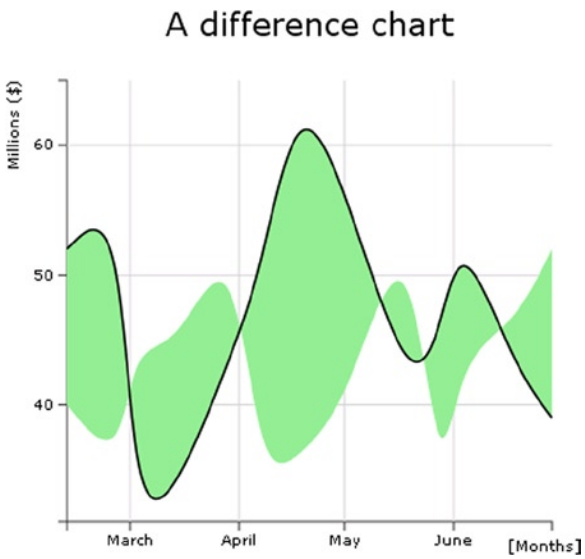


Figure 20-23. *An initial representation of the area between both trends*

But all the areas are green. Instead, you want some of these areas to be red. You need to select the areas enclosed by the income and expense lines, where the income line is above the expense line, and exclude the areas that do not correspond to this scheme. When you deal with areas, portions of which must be added or subtracted, it is necessary to introduce the **clip path** SVG element.

Clip paths are SVG elements that can be attached to previously drawn figures with a path element. The clip path describes a "window" area, which shows only in the area defined by the path. The other areas of the figure remain hidden.

Take a look at Figure 20-24. You can see that the line of income is black and thick. All green areas (light gray in the printed book version) above this line should be hidden by a clip path. But what clip path do you need? You need the clip path described by the path that delimits the lower area above the income line.

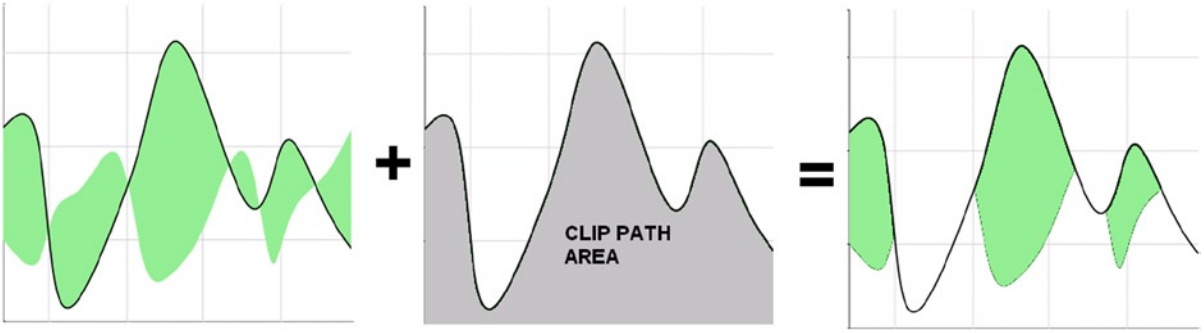


Figure 20-24. *Selection of the positive area with a clip path area*

You need to make some changes to the code, as shown in Listing 20-67.

Listing 20-67. ch20_09.html

```
d3.tsv("data_03.tsv", function(error, data) {
    ...

    svg.datum(data);

    svg.append("clipPath")
      .attr("id", "clip-below")
      .append("path")
      .attr("d", area.y0(h));

    svg.append("path")
       .attr("class", "area below")
       .attr("clip-path", "url(#clip-below)")
       .attr("d", area.y0(function(d) { return y(d.expense); }));

    svg.append("path")
       .attr("class", "line")
       .attr("d", line);

});
```

Now you need to do to same thing for the red areas (dark gray in the printed book version). Always starting from the area enclosed between the income and expense lines, you must eliminate the areas below the income line. So, as shown in Figure 20-25, you can use the clip path that describes the area above the income line as the window area.

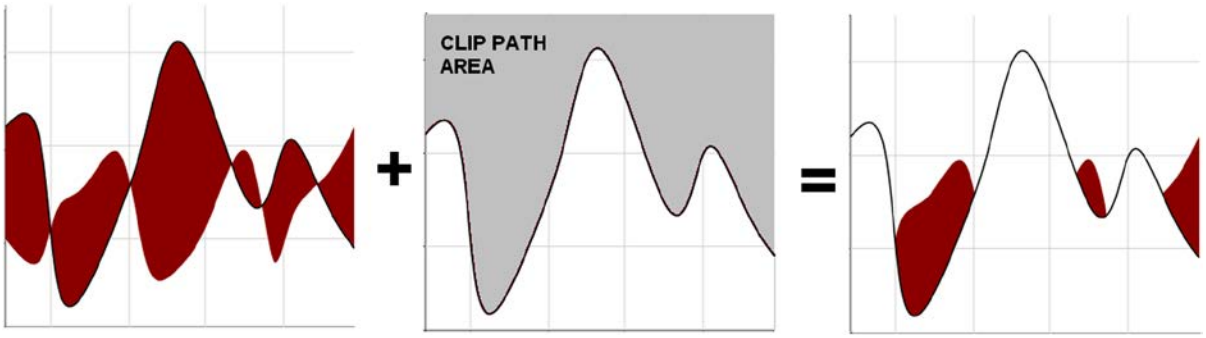


Figure 20-25. *Selection of the negative area with a clip path area*

Translating this into code, you need to add another `clipPath` to the code, as shown in Listing 20-68.

Listing 20-68. ch20_09.html

```
d3.tsv("data_03.tsv", function(error, data) {
    ...

    svg.append("path")
       .attr("class", "area below")
       .attr("clip-path", "url(#clip-below)")
       .attr("d", area.y0(function(d) { return y(d.expense); }));

    svg.append("clipPath")
       .attr("id", "clip-above")
       .append("path")
       .attr("d", area.y0(0));

    svg.append("path")
       .attr("class", "area above")
       .attr("clip-path", "url(#clip-above)")
       .attr("d", area.y0(function(d) { return y(d.expense); }));

    svg.append("path")
       .attr("class", "line")
       .attr("d", line);

});
```

In the end, both areas are drawn simultaneously, and you get the desired chart (see Figure 20-26).

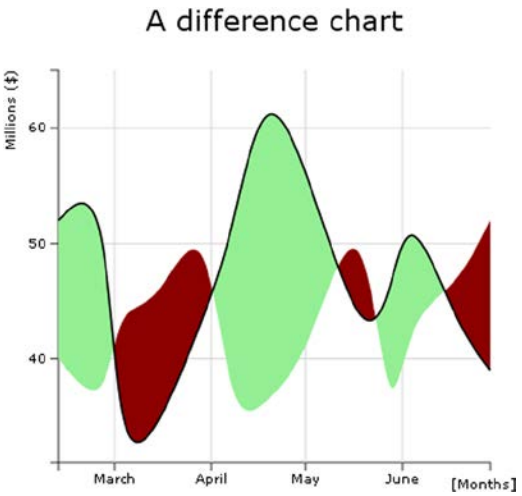


Figure 20-26. *The final representation of the difference area chart*

Summary

This chapter shows how to build the basic elements of a **line chart**, including axes, axis labels, titles, and grids. In particular you have read about the concepts of scales, domains, and ranges.

You then learned how to **read data from external files**, particularly CSV and TSV files. Furthermore, in starting to work with multiple series of data, you learned how to realize multiseries line charts, including learning about all the elements needed to complete them, such as legends.

Finally, you learned how to create a particular type of line chart: the **difference line chart**. This has helped you to understand **clip area paths**.

In the next chapter, you'll deal with **bar charts**. Exploiting all you've learned so far about D3, you'll see how it is possible to realize all the graphic components needed to build a bar chart, using only SVG elements. More specifically, you'll see how, using the same techniques, to implement all the possible types of **multiseries bar charts**, from stacked to grouped bars, both horizontally and vertically orientated.

CHAPTER 21

Bar Charts with D3

In this chapter, you will see how, using the D3 library, you can build the most commonly used type of chart: the bar chart. As a first example, you will start from a simple bar chart to practice the implementation of all the components using scalar vector graphic (SVG) elements.

Drawing a Bar Chart

In this regard, as an example to work with we choose to represent the income of some countries by vertical bars, so that we may compare them. As category labels, you will use the names of the countries themselves. Here, as you did for line charts, you decide to use an external file, such as a comma-separated values (CSV) file, which contains all the data. Your web page will then read the data contained within the file using the `d3.csv()` function. Therefore, write the data from Listing 21-1 in a file and save it as `data_04.csv`.

Listing 21-1. data_04.csv

```
country,income
France,14
Russia,22
Japan,13
South Korea,34
Argentina,28
```

Listing 21-2 shows a blank web page as a starting point for the development of your bar chart. You must remember to include the D3 library in the web page (see Appendix A for further information). If you prefer to use a content delivery network (CDN) service, you can replace the reference with this:

```
<script src="http://d3js.org/d3.v3.min.js"></script>
```

Listing 21-2. ch21_01.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<script src="../src/d3.v3.js"></script>
</head>
```

```
<body>
<script type="text/javascript">
// add the D3 code here
</script>
</body>
</html>
```

First, it is good practice to define the size of the drawing area on which you wish to represent your bar chart. The dimensions are specified by the `w` and `h` (width and height) variables, but you must also take the space for margins into account. These margin values must be subtracted from `w` and `h`, suitably restricting the area to be allocated to your chart (see Listing 21-3).

Listing 21-3. ch21_01.html

```
<script type="text/javascript">
var margin = {top: 70, right: 20, bottom: 30, left: 40},
    w = 500 - margin.left - margin.right,
    h = 350 - margin.top - margin.bottom;
var color =  d3.scale.category10();
</script>
```

Moreover, if you look at the data in the CSV file (see Listing 21-1), you will find a series of five countries with their relative values. If you want to have a color for identifying each country, it is necessary to define a color scale, and this, as you have already seen, could be done with the `category10()` function.

The next step is to define a scale on the x axis and y axis. You do not have numeric values on the x axis but string values identifying the country of origin. Thus, for this type of value, you have to define an ordinal scale as shown in Listing 21-4. In fact, the function `rangeRoundBands` divides the range passed as argument into discrete bands, which is just what you need in a bar chart. For the y axis, since it represents a variable in numerical values, you simply choose a linear scale.

Listing 21-4. ch21_01.html

```
<script type="text/javascript">
...
var color =  d3.scale.category10();
var x = d3.scale.ordinal()
    .rangeRoundBands([0, w], .1);
var y = d3.scale.linear()
    .range([h, 0]);
<script type="text/javascript">
```

Now you need to assign the two scales to the corresponding axes, using the `d3.svg.axis()` function. When you are dealing with bar charts, it is not uncommon that the values reported on the y axis are not the nominal values, but their percentage of the total value. So, you can define a percentage format through `d3.format()`, and then you assign it to tick labels on the y axis through the `tickFormat()` function (see Listing 21-5).

Listing 21-5. ch21_01.html

```
<script type="text/javascript">
...
var y = d3.scale.linear()
    .range([h, 0]);
```

```
var formatPercent = d3.format(".0%");
var xAxis = d3.svg.axis()
    .scale(x)
    .orient("bottom");
var yAxis = d3.svg.axis()
    .scale(y)
    .orient("left")
    .tickFormat(formatPercent);
<script type="text/javascript">
```

Finally, it is time to start creating SVG elements in your web page. Start with the root as shown in Listing 21-6.

Listing 21-6. ch21_01.html

```
<script type="text/javascript">
...
var yAxis = d3.svg.axis()
    .scale(y)
    .orient("left")
    .tickFormat(formatPercent);

var svg = d3.select("body").append("svg")
    .attr("width", w + margin.left + margin.right)
    .attr("height", h + margin.top + margin.bottom)
    .append("g")
    .attr("transform", "translate(" + margin.left + "," + margin.top + ")");
<script type="text/javascript">
```

And now, to access the values contained in the CSV file, you have to use the `d3.csv()` function, as you have done already, passing the file name as first argument and the iterative function on the data contained within as the second argument (see Listing 21-7).

Listing 21-7. ch21_01.html

```
<script type="text/javascript">
...
var svg = d3.select("body").append("svg")
    .attr("width", w + margin.left + margin.right)
    .attr("height", h + margin.top + margin.bottom)
    .append("g")
    .attr("transform", "translate(" + margin.left + "," + margin.top + ")");

d3.csv("data_04.csv", function(error, data) {
    var sum = 0;
    data.forEach(function(d) {
       d.income = +d.income;
       sum += d.income;
    });
    //insert here all the svg elements depending on data in the file
});
<script type="text/javascript">
```

During the scan of the values stored in the file through the `forEach()` loop, you ensure that all values of income will be read as numeric values and not as a string: this is possible by assigning every value to itself with a plus sign before it.

```
values = +values
```

And in the meantime, you also carry out the sum of all income values. This sum is necessary for you to compute the percentages. In fact, as shown in Listing 21-8, while you are defining the domains for both the axes, the "single income"/sum ratio is assigned to the y axis, thus obtaining a domain of percentages.

Listing 21-8. ch21_01.html

```
...
d3.csv("data_04.csv", function(error, data) {
    data.forEach(function(d) {
       ...
    });

    x.domain(data.map(function(d) { return d.country; }));
    y.domain([0, d3.max(data, function(d) { return d.income/sum; })]);

});
```

After setting the values on both axes, you can draw them adding the corresponding SVG elements in Listing 21-9.

Listing 21-9. ch21_01.html

```
d3.csv("data_04.csv", function(error, data) {
...
    y.domain([0, d3.max(data, function(d) { return d.income/sum; })]);

    svg.append("g")
       .attr("class", "x axis")
       .attr("transform", "translate(0," + h + ")")
       .call(xAxis);

    svg.append("g")
       .attr("class", "y axis")
       .call(yAxis);
});
```

Usually, for bar charts, the grid is required only on one axis: the one on which the numeric values are shown. Since you are working with a vertical bar chart, you draw grid lines only on the y axis (see Listing 21-10). On the other hand, grid lines are not necessary on the x axis, because you already have a kind of classification in areas of discrete values, often referred to as categories. (Even if you had continuous values to represent on the x axis, however, for the bar chart to make sense, their range on the x axis should be divided into intervals or bins. The frequency of these values at each interval is represented by the height of the bar on the y axis, and the result is a histogram.)

Listing 21-10. ch21_01.html

```
...
var yAxis = d3.svg.axis()
    .scale(y)
    .orient("left")
    .tickFormat(formatPercent);
```

```
var yGrid = d3.svg.axis()
    .scale(y)
    .orient("left")
    .ticks(5)
    .tickSize(-w, 0, 0)
    .tickFormat("");

var svg = d3.select("body").append("svg")
    .attr("width", w + margin.left + margin.right)
    .attr("height", h + margin.top + margin.bottom)
    .append("g")
    .attr("transform", "translate(" + margin.left + "," + margin.top + ")");

d3.csv("data_04.csv", function(error, data) {
    ...
    svg.append("g")
       .attr("class", "y axis")
       .call(yAxis);

    svg.append("g")
       .attr("class", "grid")
       .call(yGrid);
});
```

Since the grid lies only on the y axis, the same thing applies for the corresponding axis label. In order to separate the components of the chart between them in some way, it is good practice to define a variable for each SVG element <g> which identifies a chart component, generally with the same name you use to identify the class of the component. Thus, just as you define a `labels` variable, you also define a `title` variable. And so on, for all the other components that you intend to add.

Unlike jqPlot, it is not necessary to include a specific plug-in to rotate the axis label; rather, use one of the possible transformations which SVG provides you with, more specifically a rotation. The only thing you need to do is to pass the angle (in degrees) at which you want to rotate the SVG element. The rotation is clockwise if the passed value is positive. If, as in your case, you want to align the axis label to the y axis, then you will need to rotate it 90 degrees counterclockwise: so specify `rotate(-90)` as a transformation (see Listing 21-11). Regarding the `title` element, you place it at the top of your chart, in a central position.

Listing 21-11. ch21_01.html

```
d3.csv("data_04.csv", function(error, data) {
    ...

    svg.append("g")
      .attr("class", "grid")
      .call(yGrid);

var labels = svg.append("g")
    .attr("class","labels");

labels.append("text")
    .attr("transform", "rotate(-90)")
    .attr("y", 6)
```

```
    .attr("dy", ".71em")
    .style("text-anchor", "end")
    .text("Income [%]");

var title = svg.append("g")
    .attr("class","title");

title.append("text")
    .attr("x", (w / 2))
    .attr("y", -30 )
    .attr("text-anchor", "middle")
    .style("font-size", "22px")
    .text("My first bar chart");

});
```

Once you have defined all of the SVG components, you must not forget to specify the attributes of the CSS classes in Listing 21-12.

Listing 21-12. ch21_01.html

```
<style>
body {
    font: 14px sans-serif;
}

.axis path,
.axis line {
    fill: none;
    stroke: #000;
    shape-rendering: crispEdges;
}
.grid .tick {
    stroke: lightgrey;
    opacity: 0.7;
}
.grid path {
    stroke-width: 0;
}

.x.axis path {
    display: none;
}
</style>
```

At the end, add the SVG elements which make up your bars. Since you want to draw a bar for each set of data, here you have to take advantage of the iterative `function(error,data)` function within the `d3.csv()` function. As shown in Listing 21-13, you therefore add the `data(data)` and `enter()` functions within the function chain. Moreover, defining the `function(d)` within each `attr()` function, you can assign data values iteratively, one after the other, to the corresponding attribute. In this way, you can assign a different value (one for each row in the CSV file) to the attributes `x`, `y`, `height`, and `fill`, which affect the position, color, and size of each bar. With this mechanism each `.bar` element will reflect the data contained in one of the rows in the CSV file.

Listing 21-13. ch21_01.html

```
d3.csv("data_04.csv", function(error, data) {
    ...

    title.append("text")
      .attr("x", (w / 2))
      .attr("y", -30 )
      .attr("text-anchor", "middle")
      .style("font-size", "22px")
      .text("My first bar chart");

    svg.selectAll(".bar")
       .data(data)
       .enter().append("rect")
       .attr("class", "bar")
       .attr("x", function(d) { return x(d.country); })
       .attr("width", x.rangeBand())
       .attr("y", function(d) { return y(d.income/sum); })
       .attr("height", function(d) { return h - y(d.income/sum); })
       .attr("fill", function(d) { return color(d.country); });
});
```

At the end, all your efforts will be rewarded with the beautiful bar chart shown in Figure 21-1.

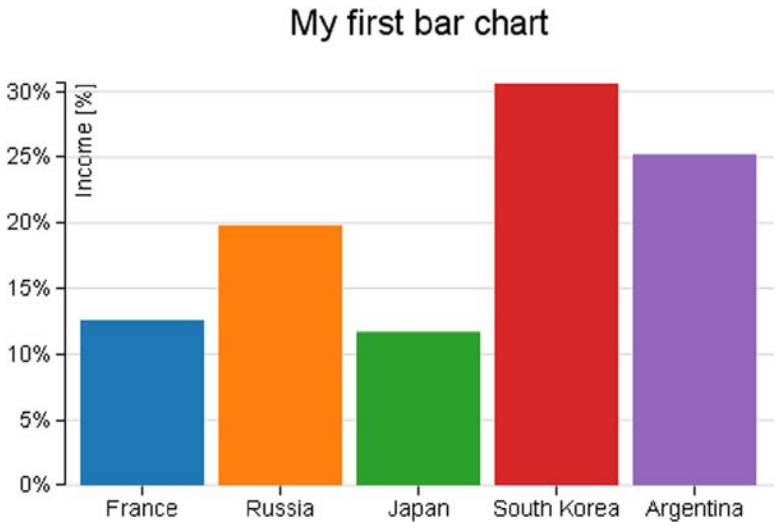


Figure 21-1. *A simple bar chart*

Drawing a Stacked Bar Chart

You have introduced the bar chart with the simplest case where you had a number of groups represented per country and a corresponding value (income). Very often, you need to represent data which are a bit more complex, for example, data in which you want to divide the total income sector by sector. In this case, you will have the income for each country

divided into various portions, each of which represents the income of a sector of production. For our example, we will use a CSV file in a way that is very similar to that used in the previous example (see Listing 21-1), but with multiple values for each country, so that you may work with multiseries bar charts. Therefore, write the data in Listing 21-14 with a text editor and save it as `data_05.csv`.

Listing 21-14. data_05.csv

```
Country,Electronics,Software,Mechanics
Germany,12,14,18
Italy,8,12,10
Spain,6,4,5
France,10,14,9
UK,7,11,9
```

Looking at the content of the file, you may notice that the columns are now four. The first column still contains the names of the nations, but now the incomes are three, each corresponding to a different sector of production: electronics, software, and mechanics. These titles are listed in the headers.

Start with the code of the previous example, making some changes and deleting some rows, until you get the code shown in Listing 21-15. The pieces of code in bold are the ones that need to be changed (the title and the CSV file)m while those that are not present must be deleted. Those of you who are starting directly from this section can easily copy the contents shown in Listing 21-15.

Listing 21-15. ch21_02.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<script src="http://d3js.org/d3.v3.js"></script>
<style>
body {
    font: 14px sans-serif;
}
.axis path,
.axis line {
    fill: none;
    stroke: #000;
    shape-rendering: crispEdges;
}
.x.axis path {
    display: none;
}
</style>
</head>
<body>
<script type="text/javascript">
var color =  d3.scale.category10();

var margin = {top: 70, right: 20, bottom: 30, left: 40},
    w = 500 - margin.left - margin.right,
    h = 350 - margin.top - margin.bottom;
```

```
var x = d3.scale.ordinal()
    .rangeRoundBands([0, w], .1);

var y = d3.scale.linear()
    .range([h, 0]);

var formatPercent = d3.format(".0%");

var xAxis = d3.svg.axis()
    .scale(x)
    .orient("bottom");

var yAxis = d3.svg.axis()
    .scale(y)
    .orient("left")
    .tickFormat(formatPercent);

var yGrid = d3.svg.axis()
    .scale(y)
    .orient("left")
    .ticks(5)
    .tickSize(-w, 0, 0)
    .tickFormat("");

var svg = d3.select("body").append("svg")
    .attr("width", w + margin.left + margin.right)
    .attr("height", h + margin.top + margin.bottom)
    .append("g")
    .attr("transform", "translate(" + margin.left + "," + margin.top + ")");

d3.csv("data_05.csv", function(error, data) {

    svg.append("g")
       .attr("class", "x axis")
       .attr("transform", "translate(0," + h + ")")
       .call(xAxis);

    svg.append("g")
       .attr("class", "y axis")
       .call(yAxis);

    svg.append("g")
       .attr("class", "grid")
       .call(yGrid);
});
var labels = svg.append("g")
    .attr("class","labels");

labels.append("text")
    .attr("transform", "rotate(-90)")
    .attr("x", 50)
    .attr("y", -20)
```

```
    .attr("dy", ".71em")
    .style("text-anchor", "end")
    .text("Income [%]");

var title = svg.append("g")
    .attr("class","title")

title.append("text")
    .attr("x", (w / 2))
    .attr("y", -30 )
    .attr("text-anchor", "middle")
    .style("font-size", "22px")
    .text("A stacked bar chart");
</script>
</body>
</html>
```

Now think of the colors and the domain you want to set. In the previous case, you drew each bar (country) with a different color. In fact, you could even give all the bars of the same color. Your approach was an optional choice, mainly due to aesthetic factors. In this case, however, using a set of different colors is necessary to distinguish the various portions that compose each bar. Thus, you will have a series of identical colors for each bar, where each color will correspond to a sector of production. A small tip: when you need a legend to identify various representations of data, you need to use a sequence of colors, and vice versa. You define the domain of colors relying on the headers of the file and deleting the first item, "Country," through a filter (see Listing 21-16).

Listing 21-16. ch21_02.html

```
d3.csv("data_05.csv", function(error, data) {

    color.domain(d3.keys(data[0]).filter(function(key) {
       return key !== "Country"; }));

    svg.append("g")
       .attr("class", "x axis")
       .attr("transform", "translate(0," + h + ")")
       .call(xAxis);

    ...
```

On the y axis you must not plot the values of the income, but their percentage of the total. In order to do this, you need to know the sum of all values of income, so with an iteration of all the data read from the file, you may obtain the sum. Again, in order to make it clear to D3 that the values in the three columns (Electronics, Mechanics, and Software) are numeric values, you must specify them explicitly in the iteration in this way:

```
values = +values;
```

In Listing 21-17, you see how the `forEach()` function iterates the values of the file and, at the same time, calculates the sum you need to obtain your percentages.

Listing 21-17. ch21_02.html

```
d3.csv("data_05.csv", function(error, data) {

    color.domain(d3.keys(data[0]).filter(function(key) {
             return key !== "Country"; }));

    var sum= 0;

    data.forEach(function(d){
       d.Electronics = +d.Electronics;
       d.Mechanics = +d.Mechanics;
       d.Software = +d.Software;
       sum = sum +d.Electronics +d.Mechanics +d.Software;
    });

    svg.append("g")
       .attr("class", "x axis")
       .attr("transform", "translate(0," + h + ")")
       .call(xAxis);

    ...
```

Now you need to create a data structure which can serve your purpose. Build an array of objects for each bar in which each object corresponds to one of the portions in which the total income is divided. Name this array "countries" and create it through an iterative function (see Listing 21-18).

Listing 21-18. ch21_02.html

```
d3.csv("data_05.csv", function(error, data) {
    ...
    data.forEach(function(d){
       d.Electronics = +d.Electronics;
       d.Mechanics = +d.Mechanics;
       d.Software = +d.Software;
       sum = sum +d.Electronics +d.Mechanics +d.Software;
    });

    data.forEach(function(d) {
       var y0 = 0;
       d.countries = color.domain().map(function(name) {
          return {name: name, y0: y0/sum, y1: (y0 += +d[name])/sum }; });
       d.total = d.countries[d.countries.length - 1].y1;
    });

    svg.append("g")
       .attr("class", "x axis")
       .attr("transform", "translate(0," + h + ")")
       .call(xAxis);

    ...
```

Using the Firebug console (see the section "Firebug and DevTool" in Chapter 1), you can directly see the internal structure of this array. Thus, add (temporarily) the call to the console to the code passing the **countries** array as an argument, as shown in Listing 21-19.

Listing 21-19. ch21_02.html

```
data.forEach(function(d) {
    var y0 = 0;
    d.countries = color.domain().map(function(name) {
       return {name: name, y0: y0/sum, y1: (y0 += +d[name])/sum }; });
    d.total = d.countries[d.countries.length - 1].y1;
    console.log(d.countries);
});
```

Figure 21-2 shows the internal structure of the `countries` array along with all its content how it is displayed by the Firebug console.

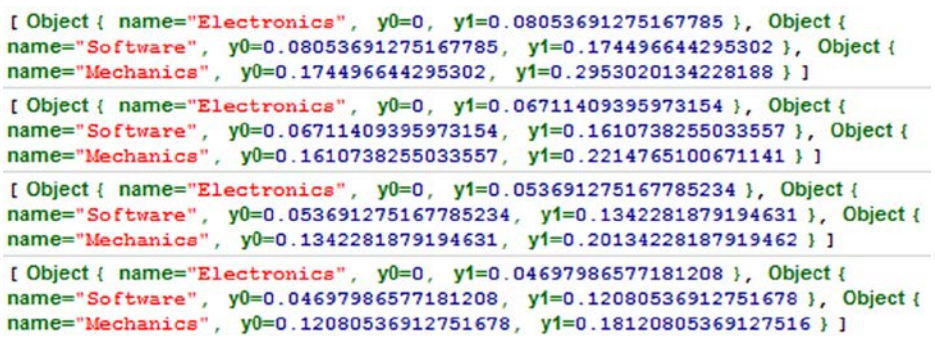


Figure 21-2. *The Firebug console shows the content and the structure of the countries array*

If you analyze in detail the first element of the array:

```
[Object { name="Electronics",  y0=0,  y1=0.08053691275167785},
Object { name="Software",  y0=0.08053691275167785,  y1=0.174496644295302},
Object { name="Mechanics",  y0=0.174496644295302,  y1=0.2953020134228188}]
```

You may notice that each element of the array is, in turn, an array containing three objects. These three objects represent the three categories (the three series of the multiseries bar chart) into which you want to split the data. The values of `y0` and `y1` are the percentages of the beginning and the end of each portion in the bar, respectively.

After you have arranged all the data you need, you can include it in the domain of x and y, as shown in Listing 21-20.

Listing 21-20. ch21_02.html

```
d3.csv("data_05.csv", function(error, data) {
    ...
  data.forEach(function(d) {
    ...
    console.log(d.countries);
  });

    x.domain(data.map(function(d) { return d.Country; }));
    y.domain([0, d3.max(data, function(d) { return d.total; })]);
```

```
    svg.append("g")
       .attr("class", "x axis")
       .attr("transform", "translate(0," + h + ")")
       .call(xAxis);

    ...
```

And then, in Listing 21-21 you start to define the `rect` elements which will constitute the bars of your chart.

Listing 21-21. ch21_02.html

```
d3.csv("data_05.csv", function(error, data) {
    ...
    svg.append("g")
      .attr("class", "grid")
      .call(yGrid);

    var country = svg.selectAll(".country")
       .data(data)
       .enter().append("g")
       .attr("class", "country")
       .attr("transform", function(d) {
          return "translate(" + x(d.Country) + ",0)"; });

    country.selectAll("rect")
       .data(function(d) { return d.countries; })
       .enter().append("rect")
       .attr("width", x.rangeBand())
       .attr("y", function(d) { return y(d.y1); })
       .attr("height", function(d) { return (y(d.y0) - y(d.y1)); })
       .style("fill", function(d) { return color(d.name); });
});
```

You have seen the internal structure of the array with the library, and since D3 always starts with basic graphics, the most complex part lies in translating the data structure into a hierarchical structure of SVG elements. The <g> tag comes to your aid to build proper hierarchical groupings. In this regard, you have to define an element <g> for each country. First, you need to use the data read from the CSV file iteratively. This can be done by passing the `data` array (the original `data` array, not the `countries` array you have just defined) as argument to the `data()` function. When all this is done, you have five new group items <g>, as five are the countries which are listed in the CSV file and five are also the bars which will be drawn. The position on the x axis of each bar should also be managed. You do not need to do any calculations to pass the right x values to `translate(x,0)` function. In fact, as shown in Figure 21-3, these values are automatically generated by D3, exploiting the fact that you have defined an ordinal scale on the x axis.

```
⊞ <g class="country" transform="translate(9,0)">
⊞ <g class="country" transform="translate(95,0)">
⊞ <g class="country" transform="translate(181,0)">
⊞ <g class="country" transform="translate(267,0)">
⊞ <g class="country" transform="translate(353,0)">
```

Figure 21-3. *Firebug shows the different translation values on the x axis that are automatically generated by the D3 library*

Within each of the group elements <g>, you must now create the <rect> elements, which will generate the colored rectangles for each portion. Furthermore, it will be necessary to ensure that the correct values are assigned to the y and height attributes, in order to properly place the rectangles one above the other, avoiding them from overlapping, and thus to obtain a single stacked bar for each country.

This time, it is the countries array which will be used, passing it as an argument to the data() function. Since it is necessary to make a further iteration for each element <g> that you have created, you will pass the iterative function(d) to the data() function as an argument. In this way, you create an iteration in another iteration: the first scans the values in data (countries); the second, inner one scans the values in the countries array (sectors of production). Thus, you assign the final percentages (y1) to the y attributes, and you assign the difference between the initial and final percentage (y0–y1) to the height attributes. The values y0 and y1 have been calculated previously when you have defined the objects contained within the countries array one by one (see Figure 21-4).

```
<g class="country" transform="translate(9,0)">
    <rect width="77" y="182" height="68" style="fill: rgb(31, 119, 180);">
    <rect width="77" y="102" height="80" style="fill: rgb(255, 127, 14);">
    <rect width="77" y="0" height="102" style="fill: rgb(44, 160, 44);">
</g>
```

Figure 21-4. *Firebug shows the different height values attributed to each rect element*

At the, end you can admire your stacked bar chart in Figure 21-5.

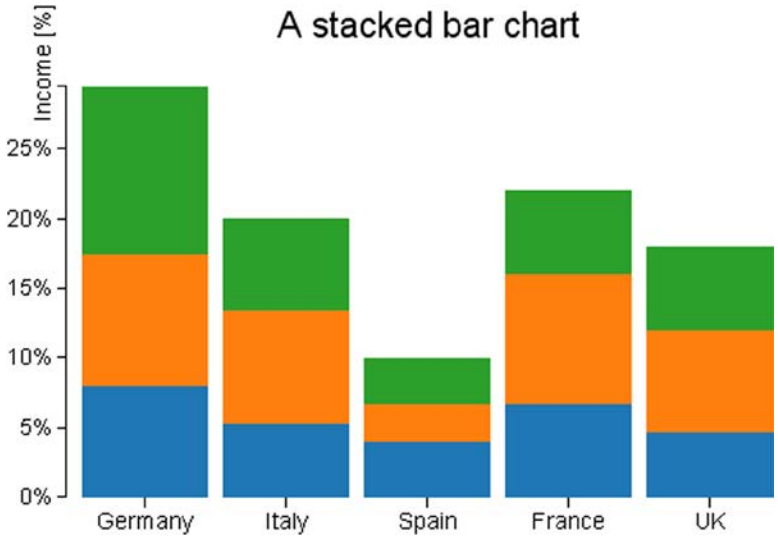


Figure 21-5. *A stacked bar chart*

Looking at your stacked bar chart, you immediately notice that something is missing. How do you recognize the sector of production, and what are their reference colors? Why not add a legend?

As you did for the other chart components, you may prefer to define a legend variable for this new component. Once you have created the group element <g>, an iteration is required also for the legend (see Listing 21-22). The iteration has to be done on the sectors of production. For each item, you need to acquire the name of the sector and the corresponding color. For this purpose, this time you will exploit the color domain which you have defined earlier: for the text element, you will use the headers in the CSV file, whereas for the color you directly assign the value of the domain.

Listing 21-22. ch21_02.html

```
d3.csv("data_05.csv", function(error, data) {
    ...
    country.selectAll("rect")
       .data(function(d) { return d.countries; })
       .enter().append("rect")
       .attr("width", x.rangeBand())
       .attr("y", function(d) { return y(d.y1); })
       .attr("height", function(d) { return  (y(d.y0) - y(d.y1)); })
       .style("fill", function(d) { return color(d.name); });

    var legend = svg.selectAll(".legend")
       .data(color.domain().slice().reverse())
       .enter().append("g")
       .attr("class", "legend")
       .attr("transform", function(d, i) {
          return "translate(0," + i * 20 + ")"; });

    legend.append("rect")
       .attr("x", w - 18)
       .attr("y", 4)
       .attr("width", 10)
       .attr("height", 10)
       .style("fill", color);

    legend.append("text")
       .attr("x", w - 24)
       .attr("y", 9)
       .attr("dy", ".35em")
       .style("text-anchor", "end")
       .text(function(d) { return d; });
});
```

Just to complete the topic of stack bar charts, using the D3 library it is possible to represent the bars in descending order, by adding the single line in Listing 21-23. Although you do not really need this feature in your case, it could be useful in certain other particular cases.

Listing 21-23. ch21_02.html

```
d3.csv("data_05.csv", function(error, data) {
    ...
    data.forEach(function(d) {
       ...
       console.log(d.countries);
    });

    data.sort(function(a, b) { return b.total - a.total; });

    x.domain(data.map(function(d) { return d.Country; }));
    ...
});
```

Figure 21-6 shows your stacked bars represented along the x axis in descending order.

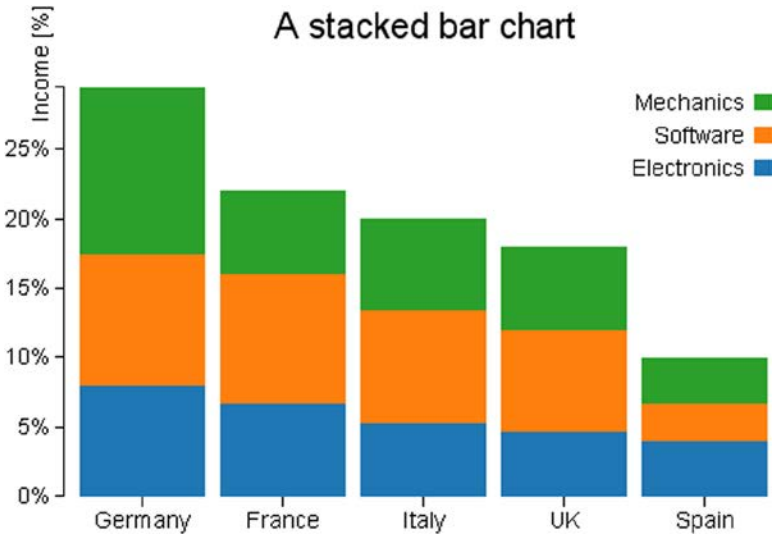


Figure 21-6. *A sorted stacked bar chart with a legend*

A Normalized Stacked Bar Chart

In this section, you will see how to convert the preceding chart in a normalized chart. By "normalized," we mean that the range of values to be displayed in the chart is converted into another target range, which often goes from 0 to 1, or 0 to 100 if you are talking about percentages (a very similar concept was treated in relation to the "Ranges, Domains, and Scales" section in Chapter 19). Therefore, if you want to compare different series covering ranges of values that are very different from each other, you need to carry out a normalization, reporting all of these intervals in percentage values between 0 and 100 (or between 0 and 1). In fact, having to compare multiple series of data, we are often interested in their relative characteristics. In our example, for instance, you might be interested in how the mechanical sector affects the economic income of a nation (normalization), and also in comparing how this influence differs from country to country (comparison between normalized values). So, in order to respond to such a demand, you can represent your stacked chart in a normalized format.

You have already reported the percentage values on the y axis; however, the percentages of each sector of production were calculated with respect to the total amount of the income of all countries. This time, the percentages will be calculated with respect to the income of each country. Thus, in this case you do not care how each individual portion partakes (in percentage) of the global income (referring to all five countries), but you care only about the percentage of income which each single sector produces in the respective country. In this case, therefore, each country will be represented by a bar at 100%. Now, there is no information about which country produces more income than others, but you are interested only in the information internal to each individual country.

All of this reasoning is important for you to understand that, although starting from the same data, you will need to choose a different type of chart depending on what you want to focus the attention of those who would be looking at the chart.

For this example, you are going to use the same file `data_05.csv` (refer to Listing 21-14); as we have just said, the incoming information is the same, but it is its interpretation which is different. In order to normalize the previous stacked bar chart, you need to effect some changes to the code. Start by extending the left and the right margins by just a few pixels as shown in Listing 21-24.

Listing 21-24. ch21_03.html

```
var margin = {top: 70, right: 70, bottom: 30, left: 50},
    w = 500 - margin.left - margin.right,
    h = 350 - margin.top - margin.bottom;
```

In Listing 21-25, inside the `d3.csv()` function you must eliminate the iterations for calculating the sum of the total income, which is no longer needed. Instead, you add a new iteration which takes the percentages referred to each country into account. Then, you must eliminate the definition of the y domain, leaving only the x domain.

Listing 21-25. ch21_03.html

```
d3.csv("data_05.csv", function(error, data) {
    color.domain(d3.keys(data[0]).filter(function(key) {
       return key !== "Country"; }));

    data.forEach(function(d) {
       var y0 = 0;
       d.countries = color.domain().map(function(name) {
          return {name: name, y0: y0, y1: y0 += +d[name]}; });
       d.countries.forEach(function(d) { d.y0 /= y0; d.y1 /= y0; });
    });

    x.domain(data.map(function(d) { return d.Country; }));

    var country = svg.selectAll(".country")
    ...
```

With this new type of chart, the y label would be covered by the bars. You must therefore delete or comment out the `rotate()` function in order to make it visible again as shown in Listing 21-26.

Listing 21-26. ch21_03.html

```
labels.append("text")
    //.attr("transform", "rotate(-90)")
    .attr("x", 50)
    .attr("y", -20)
    .attr("dy", ".71em")
    .style("text-anchor", "end")
    .text("Income [%]");
```

While you're at it, why not take the opportunity to change the title to your chart? Thus, modify the title as shown in Listing 21-27.

Listing 21-27. ch21_03.html

```
    title.append("text")
       .attr("x", (w / 2))
       .attr("y", -30 )
       .attr("text-anchor", "middle")
       .style("font-size", "22px")
       .text("A normalized stacked bar chart");
```

Even the legend is no longer required. In fact, you will replace it with another type of graphic representation which has very similar functions. Thus, you can delete the lines which define the legend in Listing 21-28 from the code.

Listing 21-28. ch21_03.html

```
var legend = svg.selectAll(".legend")
      .data(color.domain().slice().reverse())
      .enter().append("g")
      .attr("class", "legend")
      .attr("transform", function(d, i) {
               return "translate(0," + i * 20 + ")"; });

  legend.append("rect")
      .attr("x", w - 18)
      .attr("y", 4)
      .attr("width", 10)
      .attr("height", 10)
      .style("fill", color);

  legend.append("text")
      .attr("x", w - 24)
      .attr("y", 9)
      .attr("dy", ".35em")
      .style("text-anchor", "end")
      .text(function(d) { return d; });
```

Now that you have removed the legend and made the right changes, if you load the web page you get the normalized stacked bar chart in Figure 21-7.

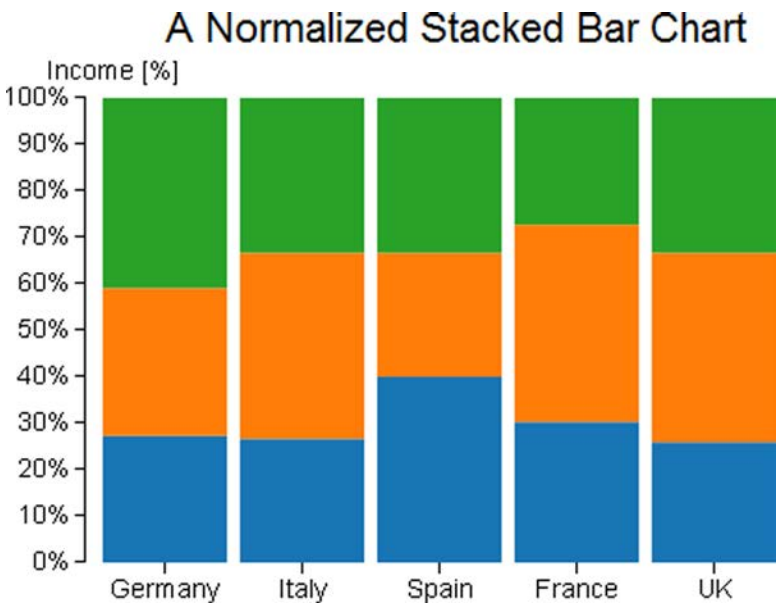


Figure 21-7. *A normalized stacked bar chart*

Without the legend, you must once again know, in some way, what the colors in the bars refer to; you are going to label the last bar on the right with labels reporting the names of the groups.

Start by adding a new style class in Listing 21-29.

Listing 21-29. ch21_03.html

```
<style>
...
.x.axis path {
  display: none;
}
.legend line {
    stroke: #000;
    shape-rendering: crispEdges;
}
</style>
```

Hence, in place of the code that you have just removed, as shown in Listing 21-28, you add the code in Listing 21-30.

Listing 21-30. ch21_03.html

```
country.selectAll("rect")
   .data(function(d) { return d.countries; })
   .enter().append("rect")
   .attr("width", x.rangeBand())
   .attr("y", function(d) { return y(d.y1); })
   .attr("height", function(d) { return (y(d.y0) - y(d.y1)); })
   .style("fill", function(d) { return color(d.name); });

var legend = svg.select(".country:last-child")
   .data(data);

legend.selectAll(".legend")
   .data(function(d) { return d.countries; })
   .enter().append("g")
   .attr("class", "legend")
   .attr("transform", function(d) {
       return "translate(" + x.rangeBand()*0.9 + "," +
          y((d.y0 + d.y1) / 2) + ")";
   });

legend.selectAll(".legend")
   .append("line")
   .attr("x2", 10);

legend.selectAll(".legend")
   .append("text")
   .attr("x", 13)
   .attr("dy", ".35em")
   .text(function(d) { return d.name; });
});
```

As you add the labels to the last bar, the SVG elements which define them must belong to the group corresponding to the last country. So, you use the `.country: last-child` selector to get the last element of the selection containing all the bars. So, the new chart will look like Figure 21-8.

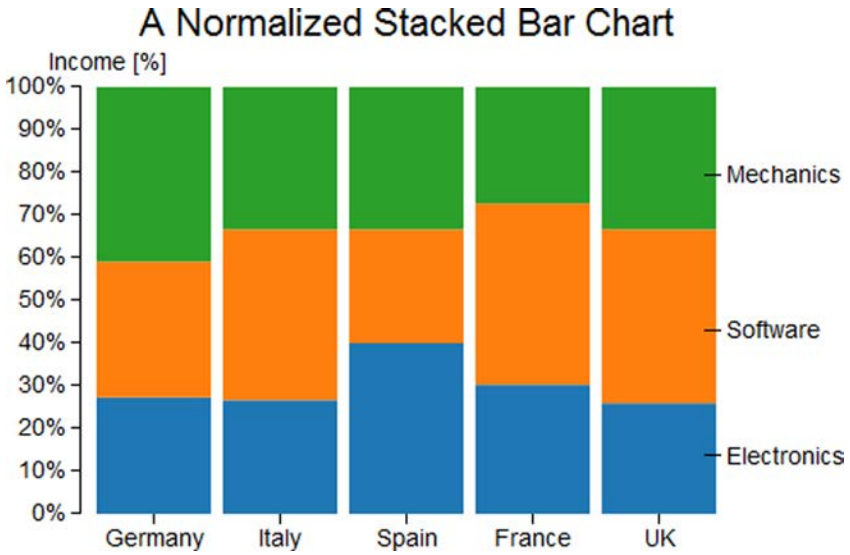


Figure 21-8. *A normalized stacked bar chart with labels as legend*

Drawing a Grouped Bar Chart

Always using the same data contained in `data_05.csv`, you can obtain another representation: a grouped bar chart. This representation is most appropriate when you want to focus on the individual income for each sector of production. In this case, you do not care in what measure the sectors partake of the total income. Thus, the percentages disappear and are replaced by y values written in the CSV file.

Listing 21-31 shows the part of code that is almost comparable to that present in other previous examples, so we will not discuss it in detail. In fact, you will use it as a starting point upon which to add other code snippets.

Listing 21-31. ch21_04.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<script src="../src/d3.v3.js"></script>
<style>
body {
  font: 14px sans-serif;
}
.axis path,
.axis line {
  fill: none;
  stroke: #000;
  shape-rendering: crispEdges;
}
```

```
.x.axis path {
  display: none;
}
</style>
</head>
<body>
<script type="text/javascript">
var color =  d3.scale.category10();

var margin = {top: 70, right: 70, bottom: 30, left: 50},
    w = 500 - margin.left - margin.right,
    h = 350 - margin.top - margin.bottom;

var yGrid = d3.svg.axis()
    .scale(y)
    .orient("left")
    .ticks(5)
    .tickSize(-w, 0, 0)
    .tickFormat("")

var svg = d3.select("body").append("svg")
    .attr("width", w + margin.left + margin.right)
    .attr("height", h + margin.top + margin.bottom)
    .append("g")
    .attr("transform", "translate(" + margin.left + "," + margin.top + ")");

d3.csv("data_05.csv", function(error, data) {

    svg.append("g")
      .attr("class", "x axis")
      .attr("transform", "translate(0," + h + ")")
      .call(xAxis);

    svg.append("g")
      .attr("class", "y axis")
      .call(yAxis)

    svg.append("g")
      .attr("class", "grid")
      .call(yGrid);
});
</script>
</body>
</html>
```

For this specific purpose, you need to define two different variables on the x axis: x0 and x1, both following an ordinal scale as shown in Listing 21-32. The x0 identifies the ordinal scale of all the groups of bars, representing a country, while x1 is the ordinal scale of each single bar within each group, representing a sector of production.

Listing 21-32. ch21_04.html

```
var margin = {top: 70, right: 70, bottom: 30, left: 50},
    w = 500 - margin.left - margin.right,
    h = 350 - margin.top - margin.bottom;

var x0 = d3.scale.ordinal()
    .rangeRoundBands([0, w], .1);
var x1 = d3.scale.ordinal();
var y = d3.scale.linear()
    .range([h, 0]);

...
```

Consequently, in the definition of the axes, you assign the *x0* to the x axis, and the y to the y axis (see Listing 21-33). Instead, the variable x1 will be used later only as a reference for the representation of the individual bar.

Listing 21-33. ch21_04.html

```
...
var y = d3.scale.linear()
    .range([h, 0]);

var xAxis = d3.svg.axis()
    .scale(x0)
    .orient("bottom");

var yAxis = d3.svg.axis()
    .scale(y)
    .orient("left");
...
```

Inside the `d3.csv()` function, you extract all the names of the sectors of production with the `keys()` function and exclude the "country" header by filtering it out from the array with the `filter()` function as shown in Listing 21-34. Here, too, you build an array of objects for each country, but the structure is slightly different. The new array looks like this:

```
[Object { name="Electronics",  value=12},
 Object { name="Software",  value=14},
 Object { name="Mechanics",  value=18}]
```

Listing 21-34. ch21_04.html

```
...
d3.csv("data_05.csv", function(error, data) {
    var sectorNames = d3.keys(data[0]).filter(function(key) {
       return key !== "Country"; });
    data.forEach(function(d) {
       d.countries = sectorNames.map(function(name) {
          return {name: name, value: +d[name]
       };
    });
    ...
});
```

Once you define the data structure, you can define the new domains as shown in Listing 21-35.

Listing 21-35. ch21_04.html

```
d3.csv("data_05.csv", function(error, data) {
    ...
    data.forEach(function(d) {
       ...
    });

    x0.domain(data.map(function(d) { return d.Country; }));
    x1.domain(sectorNames).rangeRoundBands([0, x0.rangeBand()]);
    y.domain([0, d3.max(data, function(d) {
       return d3.max(d.countries, function(d) { return d.value; });
    })]);

    svg.append("g")
       .attr("class", "x axis")
       .attr("transform", "translate(0," + h + ")")
       .call(xAxis);
...
```

As mentioned before, with `x0` you specify the ordinal domain with the names of each country. Instead, in `x1` the names of the various sectors make up the domain. Finally, in `y` the domain is defined by numerical values. Update the values passed in the iterations with the new domains (see Listing 21-36).

Listing 21-36. ch21_04.html

```
d3.csv("data_05.csv", function(error, data) {
    ...
    svg.append("g")
      .attr("class", "grid")
      .call(yGrid);

    var country = svg.selectAll(".country")
       .data(data)
       .enter().append("g")
       .attr("class", "country")
       .attr("transform", function(d) {
          return "translate(" + x0(d.Country) + ",0)";
       });

    country.selectAll("rect")
       .data(function(d) { return d.countries; })
       .enter().append("rect")
       .attr("width", x1.rangeBand())
       .attr("x", function(d) { return x1(d.name); })
       .attr("y", function(d) { return y(d.value); })
       .attr("height", function(d) { return h - y(d.value); })
       .style("fill", function(d) { return color(d.name); });
});
```

Then, externally to the `csv()` function you can define the SVG element which will represent the axis label on the y axis, as shown in Listing 21-37. It does not need to be defined within the `csv()` function, since it is independent from the data contained in the CSV file.

Listing 21-37. ch21_04.html

```
d3.csv("data_05.csv", function(error, data) {
    ...
});

var labels = svg.append("g")
    .attr("class","labels")

labels.append("text")
    .attr("transform", "rotate(-90)")
    .attr("y", 5)
    .attr("dy", ".71em")
    .style("text-anchor", "end")
    .text("Income");
</script>
```

One last thing . . . you need to add an appropriate title to the chart as shown in Listing 21-38.

Listing 21-38. ch21_04.html

```
labels.append("text")
    ...
    .text("Income");

var title = svg.append("g")
    .attr("class","title")

title.append("text")
    .attr("x", (w / 2))
    .attr("y", -30 )
    .attr("text-anchor", "middle")
    .style("font-size", "22px")
    .text("A grouped bar chart");
</script>
```

And Figure 21-9 is the result.

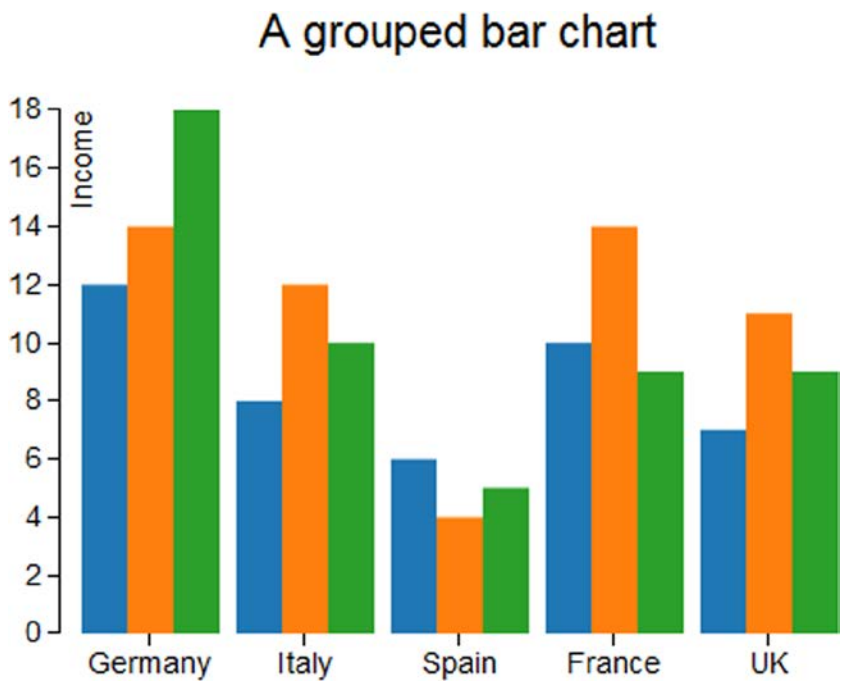


Figure 21-9. *A grouped bar chart*

In the previous case, with the normalized bar chart, you looked at an alternative way to represent a legend. You have built this legend by putting some labels which report the series name on the last bar (refer to Figure 21-8). Actually you made use of point labels. These labels can contain any text and are directly connected to a single value in the chart. At this point, introduce point labels. You will place them at the top of each bar, showing the numerical value expressed by that bar. This greatly increases the readability of each type of chart.

As you have done for any other chart component, having defined the *`PointLabels`* variable, you use it in order to assign the chain of functions applied to the corresponding selection. Also, for this type of component, which has specific values for individual data, you make use of an iteration for the data contained in the CSV file. The data on which you want to iterate are the same data you used for the bars. You therefore pass the same iterative `function(d)` to the `data()` function as argument (see Listing 21-39). In order to draw the data on top of the bars, you will apply a `translate()` transformation for each *`PointLabel`*.

Listing 21-39. ch21_04.html

```
d3.csv("data_05.csv", function(error, data) {
    ...
country.selectAll("rect")
      ...
      .attr("height", function(d) { return h - y(d.value); })
      .style("fill", function(d) { return color(d.name); });

var pointlabels = country.selectAll(".pointlabels")
    .data(function(d) { return d.countries; })
    .enter().append("g")
    .attr("class", "pointlabels")
```

```
    .attr("transform", function(d) {
       return "translate(" + x1(d.name) + "," + y(d.value) + ")";
    })
    .append("text")
    .attr("dy", "-0.3em")
    .attr("x", x1.rangeBand()/2)
    .attr("text-anchor", "middle")
    .text(function(d) { return d.value; });
    ...
});
```

And finally, there is nothing left to do but to add a legend, grouped in the classic format, to the chart (see Listing 21-40).

Listing 21-40. ch21_04.html

```
d3.csv("data_05.csv", function(error, data) {
    ...
    pointlabels.append("text")
       ...
       .text(function(d) { return d.value; });

    var legend = svg.selectAll(".legend")
       .data(color.domain().slice().reverse())
       .enter().append("g")
       .attr("class", "legend")
       .attr("transform", function(d, i) {
          return "translate(0," + i * 20 + ")";
       });

    legend.append("rect")
       .attr("x", w - 18)
       .attr("y", 4)
       .attr("width", 10)
       .attr("height", 10)
       .style("fill", color);

    legend.append("text")
       .attr("x", w - 24)
       .attr("y", 9)
       .attr("dy", ".35em")
       .style("text-anchor", "end")
       .text(function(d) { return d; });
});
```

Figure 21-10 shows the new chart with point labels and a segments legend.

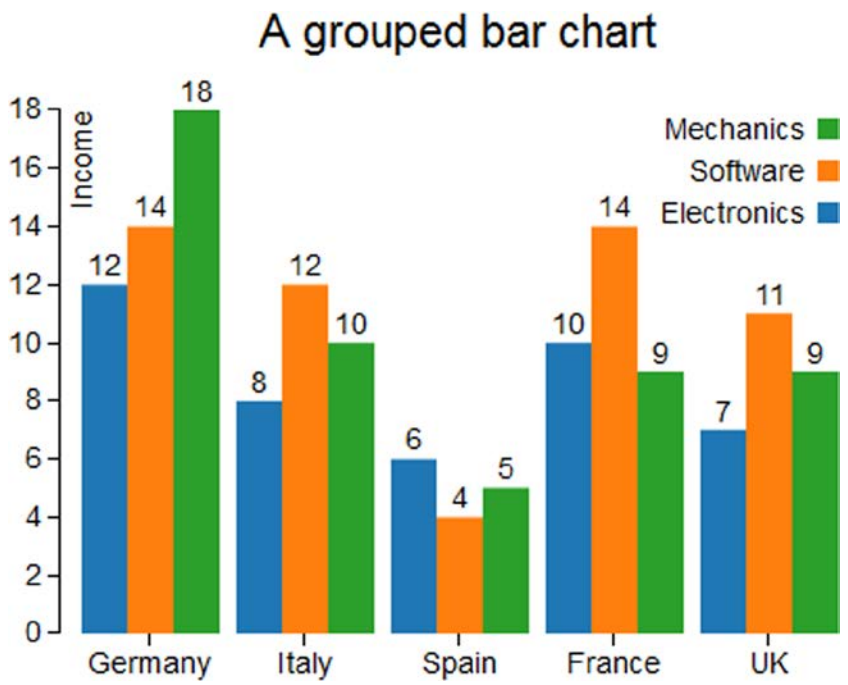


Figure 21-10. *A grouped bar chart reporting the values above each bar*

Horizontal Bar Chart with Negative Values

So far you have used only positive values, but what if you have both positive and negative values? How can you represent them in a bar chart? Take, for example, this sequence of values containing both positive and negative values (see Listing 21-41).

Listing 21-41. ch21_05.html

```
var data = [4, 3, 1, -7, -10, -7, 1, 5, 7, -3, -5, -12, -7, -11, 3, 7, 8, -1];
```

Before analyzing the data to be displayed, start adding margins to your charts as shown in Listing 21-42.

Listing 21-42. ch21_05.html

```
var data = [4, 3, 1, -7, -10, -7, 1, 5, 7, -3, -5, -12, -7, -11, 3, 7, 8, -1];

var margin = {top: 30, right: 10, bottom: 10, left: 30},
    w = 700 - margin.left - margin.right,
    h = 400 - margin.top - margin.bottom;
```

In this particular case, you will make use of horizontal bars where the values in the input array will be represented on the x axis, with the value 0 in the middle. In order to achieve this, it is first necessary to find the maximum value in absolute terms (both negative and positive). You then create the x variable on a linear scale, while the y variable is assigned to an ordinal scale containing the sequence in which data are placed in the input array (see Listing 21-43).

Listing 21-43. ch21_05.html

```
...
var margin = {top: 30, right: 10, bottom: 10, left: 30},
    w = 700 - margin.left - margin.right,
    h = 400 - margin.top - margin.bottom;

var xMax = Math.max(-d3.min(data), d3.max(data));

var x = d3.scale.linear()
    .domain([-xMax, xMax])
    .range([0, w])
    .nice();

var y = d3.scale.ordinal()
    .domain(d3.range(data.length))
    .rangeRoundBands([0, h], .2);
```

In Listing 21-44, you assign the two scales to the corresponding x axis and y axis. This time, the x axis will be drawn in the upper part of the chart while the y axis will be oriented downwards (the y values are growing downwards).

Listing 21-44. ch21_05.html

```
var y = d3.scale.ordinal()
    .domain(d3.range(data.length))
    .rangeRoundBands([0, h], .2);

var xAxis = d3.svg.axis()
    .scale(x)
    .orient("top");
var yAxis = d3.svg.axis()
    .scale(y)
    .orient("left");
```

At this point, there is nothing left to do but to begin to implement the drawing area. Create the root `<svg>` element, assigning the margins that have been previously defined. Then, you define the x axis and y axis (see Listing 21-45).

Listing 21-45. ch21_05.html

```
var yAxis = d3.svg.axis()
    .scale(y)
    .orient("left")

var svg = d3.select("body").append("svg")
    .attr("width", w + margin.left + margin.right)
    .attr("height", h + margin.top + margin.bottom)
    .append("g")
    .attr("transform", "translate(" + margin.left + "," + margin.top + ")");
```

```
svg.append("g")
    .attr("class", "x axis")
    .call(xAxis);

svg.append("g")
    .attr("class", "y axis")
    .attr("transform", "translate("+x(0)+",0)")
    .call(yAxis);
```

Finally, you need to insert a <rect> element for each bar to be represented, being careful to divide the bars into two distinct groups: negative bars and positive bars (see Listing 21-46). These two categories must be distinguished in order to set their attributes separately, with a CSS style class (e.g., color).

Listing 21-46. ch21_05.html

```
svg.append("g")
      .attr("class", "y axis")
      .attr("transform", "translate("+x(0)+",0)")
      .call(yAxis);

svg.selectAll(".bar")
    .data(data)
    .enter().append("rect")
    .attr("class", function(d) {
       return d < 0 ? "bar negative" : "bar positive";
    })
    .attr("x", function(d) { return x(Math.min(0, d)); })
    .attr("y", function(d, i) { return y(i); })
    .attr("width", function(d) { return Math.abs(x(d) - x(0)); })
    .attr("height", y.rangeBand());
```

In fact, if you analyze the structure with Firebug in Figure 21-11, you will see that the iteration has created two different types of bars within the same group, recognizable by the characterization of the class name "bar positive" and "bar negative." Through these two different names, you apply two different CSS styles in order to distinguish the bars with negative values from those with positive values.

```
<g class="y axis" transform="translate(330,0)">
  <rect class="bar positive" x="330" y="11" width="110" height="15">
  <rect class="bar positive" x="330" y="30" width="82.5" height="15">
  <rect class="bar positive" x="330" y="49" width="27.5" height="15">
  <rect class="bar negative" x="137.5" y="68" width="192.5" height="15">
  <rect class="bar negative" x="55" y="87" width="275" height="15">
  <rect class="bar negative" x="137.5" y="106" width="192.5" height="15">
  <rect class="bar positive" x="330" y="125" width="27.5" height="15">
  <rect class="bar positive" x="330" y="144" width="137.49999999999994" height="15">
```

Figure 21-11. *Firebug shows how it is possible to distinguish the positive from the negative bars, indicating the distinction in the class of each rect element*

According to what we have just said, you set the style class attributes for negative and positive bars as in Listing 21-47.

Listing 21-47. ch21_05.html

```
<style>
.bar.positive {
    fill: red;
    stroke: darkred;
}
.bar.negative {
    fill: lightblue;
    stroke: blue;
}
.axis path,
.axis line {
    fill: none;
    stroke: #000;
}
body {
    font: 14px sans-serif;
}
</style>
```

At the end, you get the chart in Figure 21-12 with red bars for positive values and blue bars for negative values.

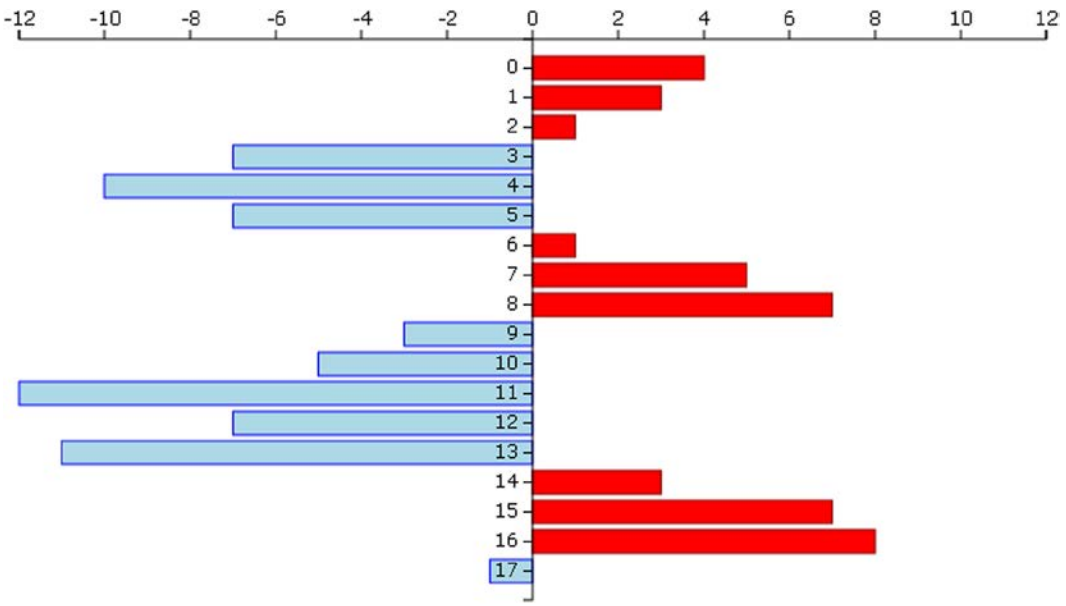


Figure 21-12. *A horizontal bar chart*

Summary

In this chapter, you have covered almost all of the fundamental aspects related to the implementation of a bar chart, the same type of chart which was developed in the first section of the book using the jqPlot library. Here, you made use of the D3 library. Thus, you saw how you can realize a simple bar chart element by element; then you moved on to the various cases of stacked bar charts and grouped bar charts, to finally look at a most peculiar case: a horizontal bar chart which portrays negative values.

In the next chapter, you will continue with the same approach: you will learn how to implement pie charts in a way similar to that used when working with jqPlot, but this time you will be using the D3 library.

CHAPTER 22

Pie Charts with D3

In the previous chapter, you have just seen how bar charts represent a certain category of data. You have also seen that starting from the same data structure, depending on your intentions you could choose one type of chart rather than another in order to accentuate particular aspects of the data. For instance, in choosing a normalized stacked bar chart, you wanted to focus on the percentage of income that each sector produces in its country.

Very often, such data represented by bar charts can also be represented using pie charts. In this chapter, you will learn how to create even this type of chart using the D3 library. Given that this library does not provide the graphics which are already implemented, as jqPlot does, but requires the user to build them using basic scalar vector graphics (SVG) elements, you will start by looking at how to build arcs and circular sectors. In fact, as with rectangles for bar charts and lines for line charts, these shapes are of fundamental importance if you are to realize pie charts (using circular sectors) or donut charts (using arcs). After you have implemented a classic example of a pie chart, we will deepen the topic further, by creating some variations. In the second part of the chapter, you will tackle donut charts, managing multiple series of data that are read from a comma-separated values (CSV) file.

Finally, we will close the chapter with a chart that we have not dealt with yet: the polar area diagram. This type of chart is a further evolution of a pie chart, in which the slices are no longer enclosed in a circle, but all have different radii. With polar area diagram, the information will no longer be expressed only by the angle that a slice occupies but also by its radius.

The Basic Pie Charts

To better highlight the parallels between bar charts and pie charts, in this example you will use the same CSV file that you used to create a basic bar chart (see the "Drawing a bar chart" section in Chapter 21). Thus, in this section, your purpose will be to implement the corresponding pie chart using the same data. In order to do this, before you start "baking" pies and donuts, you must first obtain "baking trays" of the right shape. The D3 library also allows you to represent curved shapes such as arches and circular sectors, although there actually are no such SVG elements. In fact, as you will soon see, thanks to some of its methods D3 can handle arcs and sectors as it handles other real SVG elements (rectangles, circles, lines, etc.). Once you are confident with the realization of these elements, your work in the creation of a basic pie chart will be almost complete. In the second part of this section, you will produce some variations on the theme, playing mainly with shape borders and colors in general.

Drawing a Basic Pie Chart

Turn your attention again to data contained in the CSV file named `data_04.csv` (see Listing 22-1).

Listing 22-1. data_04.csv

```
country,income
France,14
Russia,22
Japan,13
South Korea,34
Argentina,28
```

Now, we will demonstrate how these data fit well in a pie chart representation. First, in Listing 22-2, the drawing area and margins are defined.

Listing 22-2. ch22_01a.html

```
var margin = {top: 70, right: 20, bottom: 30, left: 40},
    w = 500 - margin.left - margin.right,
    h = 400 - margin.top - margin.bottom;
```

Even for pie charts, you need to use a sequence of colors to differentiate the slices between them. Generally, it is usual to use the `category10()` function to create a domain of colors, and that is what you have done so far. You could do the same thing in this example, but this is not always required. We thus take advantage of this example to see how it is possible to pass a sequence of custom colors. Create a customized example by defining the colors to your liking, one by one, as shown in Listing 22-3.

Listing 22-3. ch22_01a.html

```
var margin = {top: 70, right: 20, bottom: 30, left: 40},
    w = 500 - margin.left - margin.right,
    h = 400 - margin.top - margin.bottom;

var color = d3.scale.ordinal()
    .range(["#ffc87c", "#ffeba8", "#f3b080", "#916800", "#dda66b"]);
```

Whereas previously you had bars built with `rect` elements, now you have to deal with the sections of a circle. Thus, you are dealing with circles, angles, arches, radii, etc. In D3 there is a whole set of tools which allow you to work with these kinds of objects, making your work with pie charts easier.

To express the slices of a pie chart (circle sectors), D3 provides you with a function: `d3.svg.arc()`. This function actually defines the arches. By the term "arc," we mean a particular geometric surface bound by an angle and by two circles, one with a smaller radius (inner radius) and the other with a larger radius (outer radius). The circular sector, i.e., the slice of a pie chart, is nothing more than an arc with an inner radius equal to 0 (see Figure 22-1).

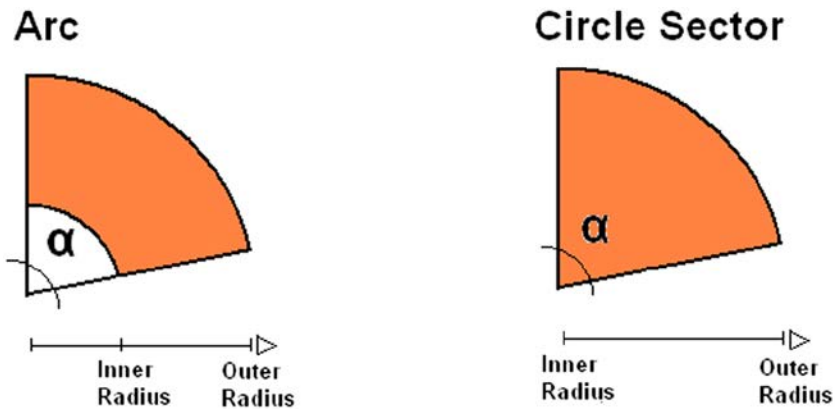


Figure 22-1. *By increasing the inner radius, it is possible to switch from a circle sector to an arc*

First, you calculate a radius which is concordant to the size of the drawing area. Then, according to this range, you delimit the outer radius and inner radius, which in this case is 0 (see Listing 22-4).

Listing 22-4. ch22_01a.html

```
...
var color = d3.scale.ordinal()
    .range(["#ffc87c", "#ffeba8", "#f3b080", "#916800", "#dda66b"]);

var radius = Math.min(w, h) / 2;
var arc = d3.svg.arc()
    .outerRadius(radius)
    .innerRadius(0);
```

D3 also provides a function to define the pie chart: the `d3.layout.pie()` function. This function builds a layout that allows you to compute the start and end angles of an arc in a very easy way. It is not mandatory to use such a function, but the pie layout automatically converts an array of data into an array of objects. Thus, define a pie with an iterative function on income values as shown in Listing 22-5.

Listing 22-5. ch22_01a.html

```
...
var arc = d3.svg.arc()
    .outerRadius(radius)
    .innerRadius(0);

var pie = d3.layout.pie()
    .sort(null)
    .value(function(d) { return d.income; });
```

Now, as seen in Listing 22-6, you insert the root element `<svg>`, assigning the correct dimensions and the appropriate `translate()` transformation.

Listing 22-6. ch22_01a.html

```
...
var pie = d3.layout.pie()
    .sort(null)
    .value(function(d) { return d.income; });

var svg = d3.select("body").append("svg")
    .attr("width", w + margin.left + margin.right)
    .attr("height", h + margin.top + margin.bottom)
    .append("g")
    .attr("transform", "translate(" +(w / 2 + margin.left) +
       "," + (h / 2 + margin.top) + ")");
```

Next, for the reading of data in CSV files, you use the `d3.csv()` function, as always. Here too you must ensure that the income is interpreted in numeric values and not as strings. Then, you write the iteration with the `forEach()` function and assign the values of income with the sign '+' beside them, as shown in Listing 22-7.

Listing 22-7. ch22_01a.html

```
...
    .append("g")
    .attr("transform", "translate(" +(w/2+margin.left)+
        "," +(h/2+margin.top)+ ")");

d3.csv("data_04.csv", function(error, data) {

    data.forEach(function(d) {
       d.income = +d.income;
    });

});
```

It is now time to add an `<arc>` item, but this element does not exist as an SVG element. In fact, what you use here really is a `<path>` element which describes the shape of the arc. It is D3 itself which builds the corresponding path thanks to the `pie()` and `arc()` functions. This spares you a job which is really too complex. You are left only with the task of defining these elements as if they were `<arc>` elements (see Listing 22-8).

Listing 22-8. ch22_01a.html

```
d3.csv("data_04.csv", function(error, data) {

    data.forEach(function(d) {
       d.income = +d.income;
    });

    var g = svg.selectAll(".arc")
       .data(pie(data))
       .enter().append("g")
       .attr("class", "arc");
```

```
    g.append("path")
       .attr("d", arc)
       .style("fill", function(d) { return color(d.data.country); });

});
```

If you analyze the SVG structure with Firebug, you can see in Figure 22-2 that the arc paths are created automatically, and that you have a <g> element for each slice.

Figure 22-2. *With Firebug, you can see how the D3 library automatically builds the arc element*

Moreover, it is necessary to add an indicative label to each slice so that you can understand which country it relates to, as shown in Listing 22-9. Notice the `arc.centroid()` function. This function computes the centroid of the arc. The centroid is defined as the midpoint between the inner and outer radius and the start and end angle. Thus, the label text appears perfectly in the middle of every slice.

Listing 22-9. ch22_01a.html

```
d3.csv("data_04.csv", function(error, data) {
    ...
    g.append("path")
       .attr("d", arc)
       .style("fill", function(d) { return color(d.data.country); });

    g.append("text")
       .attr("transform", function(d) {
          return "translate(" + arc.centroid(d) + ")"; })
       .style("text-anchor", "middle")
       .text(function(d) { return d.data.country; });

});
```

Even for pie charts, it is good practice to add a title at the top and in a central position (see Listing 22-10).

Listing 22-10. ch22_01a.html

```
d3.csv("data_04.csv", function(error, data) {
    ...
    g.append("text")
      .attr("transform", function(d) {
         return "translate(" + arc.centroid(d) + ")"; })
      .style("text-anchor", "middle")
      .text(function(d) { return d.data.country; });

    var title = d3.select("svg").append("g")
       .attr("transform", "translate(" +margin.left+ "," +margin.top+ ")")
       .attr("class","title")

    title.append("text")
       .attr("x", (w / 2))
       .attr("y", -30 )
       .attr("text-anchor", "middle")
       .style("font-size", "22px")
       .text("My first pie chart");
});
```

As for the CSS class attributes, you can add the definitions in Listing 22-11.

Listing 22-11. ch22_01a.html

```
<style>
body {
    font: 16px sans-serif;
}

.arc path {
    stroke: #000;
}
</style>
```

At the end, you get your first pie chart using the D3 library, as shown in Figure 22-3.

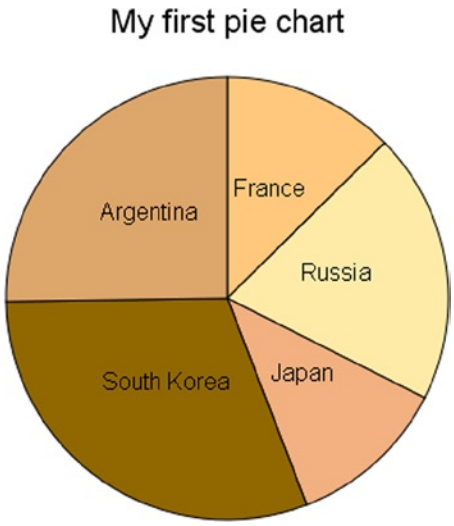


Figure 22-3. *A simple pie chart*

Some Variations on Pie Charts

Now, you will effect some changes to the basic pie chart that you have just created, illustrating some of the infinite possibilities of variations on the theme that you can obtain:

- working on color sequences;
- sorting the slices in a pie chart;
- adding spaces between the slices;
- representing the slices only with outlines;
- combining all of the above.

Working on Color Sequences

In the previous example, we defined the colors in the scale, and in the preceding examples we used the `category10()` function. There are other already defined categorical color scales: `category20()`, `category20b()`, and `category20c()`. Apply them to your pie chart, just to see how they affect its appearance. Listing 22-12 shows a case in which you use the `category10()` function. For the other categories, you only need to replace this function with others.

Listing 22-12. ch22_01b.html

```
var color = d3.scale.category10();
```

Figure 22-4 shows the color variations among the scales (reflected in different grayscale tones in print). The category10() and category20() functions generate a scale with alternating colors; instead, category20b() and category 20c() generate a scale with a slow gradation of colors.

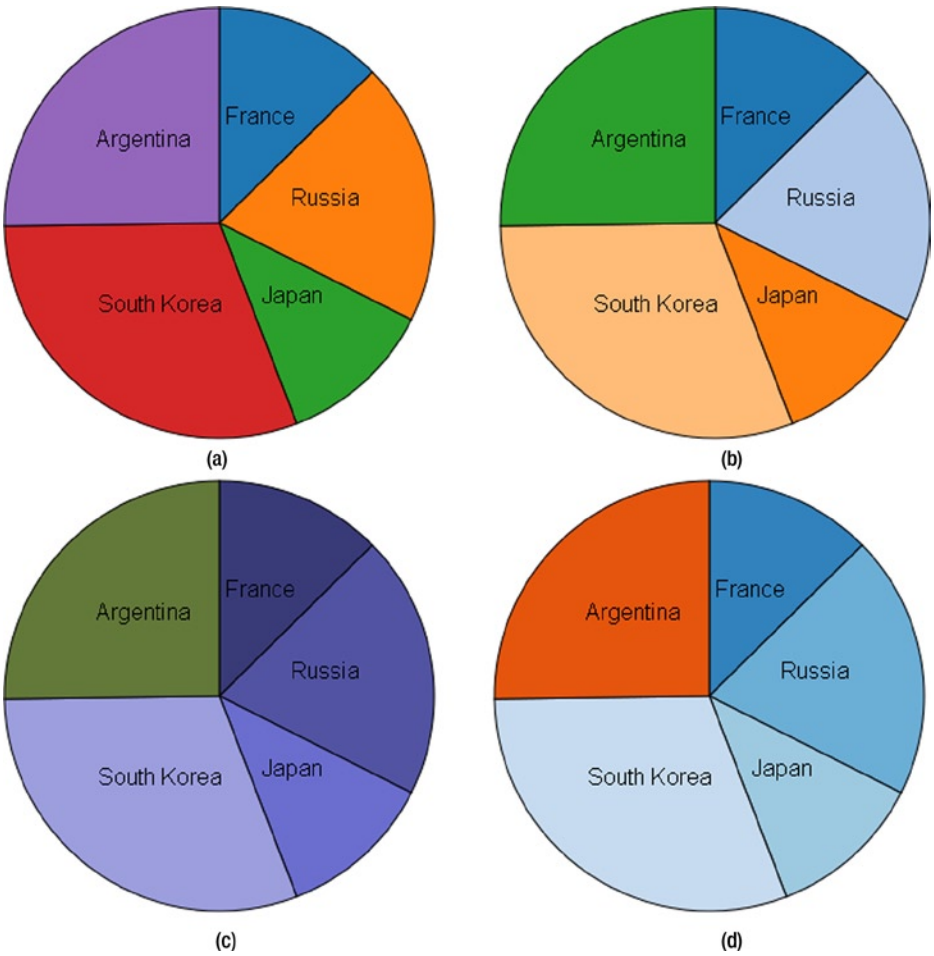


Figure 22-4. *Different color sequences: a) category10, b) category20, c) category20b, d) category20c*

Sorting the Slices in a Pie Chart

Another thing to notice is that, by default, the pie chart in D3 undergoes an implicit sorting. Thus, in case you had not made the request explicitly by passing `null` to the `sort()` function, as shown in Listing 22-13.

Listing 22-13. ch22_01c.html

```
var pie = d3.layout.pie()
    //.sort(null)
    .value(function(d) { return d.income; });
```

Then, the pie chart would have had looked different, as shown in Figure 22-5.

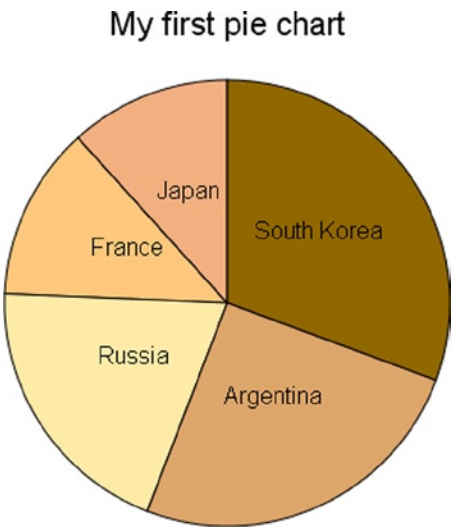


Figure 22-5. *A simple pie chart with sorted slices*

In a pie chart, the first slice is the greatest, and then the other slices are gradually added in descending order.

Adding Spaces Between the Slices

Often, the slices are shown spaced out between them, and this can be achieved very easily. You just need to make some changes to the CSS style class for the `path` elements as shown in Listing 22-14.

Listing 22-14. ch22_01d.html

```
.arc path {
    stroke: #fff;
    stroke-width: 4;
}
```

Figure 22-6 shows how the pie chart assumes a more pleasing appearance when the slices are spaced apart by a white gap.

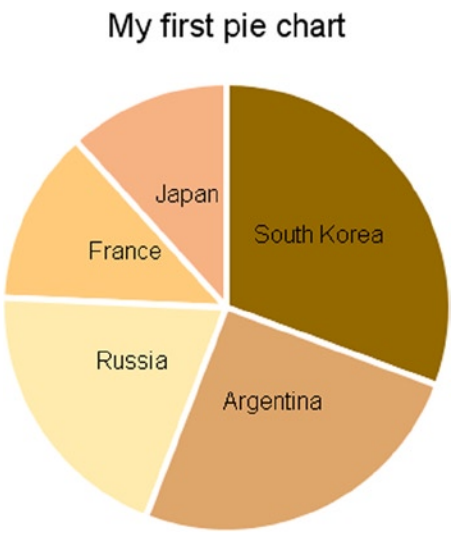


Figure 22-6. *The slices are separated by a white space*

Representing the Slices Only with Outlines

It is a little more complex to draw your pie chart with slices which are bounded only by colored borders and empty inside. You have seen similar cases with jqPlot. Change the CSS style class, as shown in Listing 22-15.

Listing 22-15. ch22_01e.html

```
.arc path {
    fill: none;
    stroke-width: 6;
}
```

In fact, this time you do not want to fill the slices with specific colors, but you want the edges, defining them, to be colored with specific colors. So you need to replace the `fill` with the `stroke` attribute in the style definition of the SVG element. Now it is the line which is colored with the indicative color. But you need to make another change, which is a little bit more complex to understand.

You are using the borders of every slice to specify the colored part, but they are actually overlapped. So, the following color covers the previous one partially and it is not so neat to have all the slices attached. It would be better to add a small gap. This could be done very easily, simply by applying a translation for each slice. Every slice should be driven off the center by a small distance, in a centrifugal direction. Thus, the translation is different for each slice and here you exploit the capabilities of the `centroid()` function, which gives you the direction (x and y coordinates) of the translation (see Listing 22-16).

Listing 22-16. ch22_01e.html

```
var g = svg.selectAll(".arc")
    .data(pie(data))
    .enter().append("g")
    .attr("class", "arc")
```

```
    .attr("transform", function(d) {
       a = arc.centroid(d)[0]/6;
       b = arc.centroid(d)[1]/6;
       return "translate(" + a +","+b + ")";
    })

g.append("path")
    .attr("d", arc)
    .style("stroke", function(d) { return color(d.data.country); });
```

Figure 22-7 illustrates how these changes affect the pie chart.

Figure 22-7. *A pie chart with unfilled slices*

Mixing All of This

But it does not end here. You can create a middle solution between these last two pie charts: get the slices with more intensely colored edges and fill them with a lighter color. It is enough to define two identical but differently colored paths, as shown in Listing 22-17. The first will have a uniform color which is slightly faint, while the second will have only colored edges and the inside will be white.

Listing 22-17. ch22_01f.html

```
var g = svg.selectAll(".arc")
    .data(pie(data))
    .enter().append("g")
    .attr("class", "arc")
    .attr("transform", function(d) {
       a = arc.centroid(d)[0]/6;
       b = arc.centroid(d)[1]/6;
       return "translate(" + a +","+b + ")";
    })
```

```
g.append("path")
    .attr("d", arc)
    .style("fill", function(d) { return color(d.data.country); })
    .attr('opacity', 0.5);

g.append("path")
    .attr("d", arc)
    .style("stroke", function(d) { return color(d.data.country); });
```

Figure 22-8 shows the spaced pie chart with two paths that color the slices and their borders.

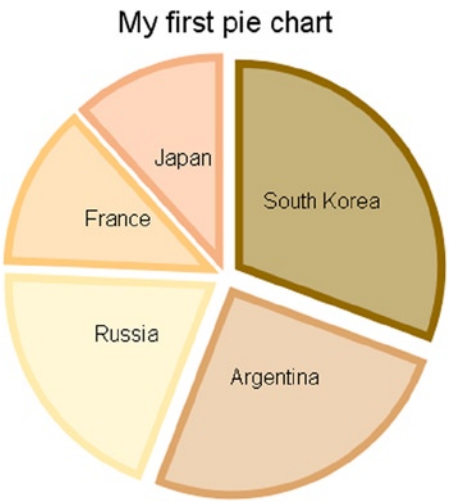


Figure 22-8. *A different way to color the slices in a pie chart*

Donut Charts

As the pie chart is to the bar chart, so the donut chart is to the multiseries bar chart. In fact, when you have multiple sets of values, you would have to represent them with a pie chart for each series. If you use donuts charts instead, you can represent them all together and also compare them in a single chart (see Figure 22-9).

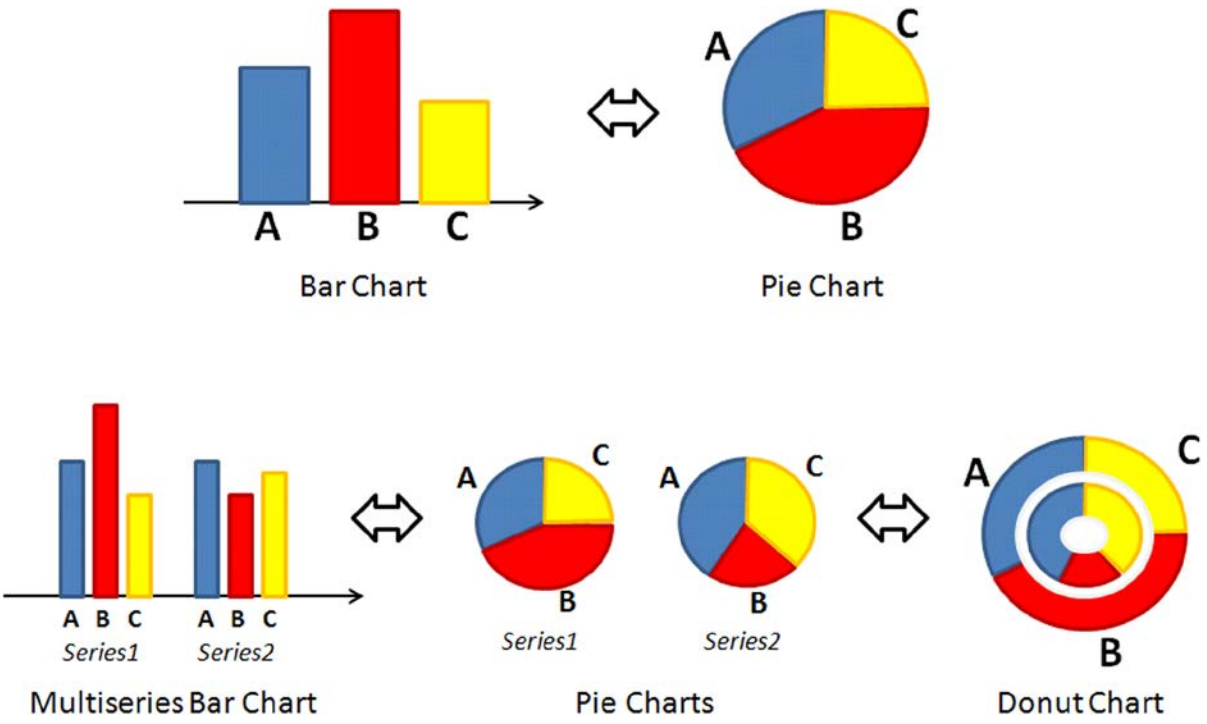


Figure 22-9. *A diagram representing the parallelism between pie charts and bar charts both with one and with multiple series of data*

Begin by writing the code in Listing 22-18; we will not provide any explanation because it is identical to the previous example.

Listing 22-18. ch22_02.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<script src="http://d3js.org/d3.v3.js"></script>
<style>
body {
    font: 16px sans-serif;
}
.arc path {
    stroke: #000;
}
</style>
</head>
<body>
<script type="text/javascript">
var margin = {top: 70, right: 20, bottom: 30, left: 40},
    w = 500 - margin.left - margin.right,
    h = 400 - margin.top - margin.bottom;
```

```
var color = d3.scale.ordinal()
    .range(["#ffc87c", "#ffeba8", "#f3b080", "#916800", "#dda66b"]);

var radius = Math.min(w, h) / 2;

var svg = d3.select("body").append("svg")
    .attr("width", w + margin.left + margin.right)
    .attr("height", h + margin.top + margin.bottom)
    .append("g")
    .attr("transform", "translate(" +(w/2+margin.left)+
       "," +(h/2+margin.top)+ ")");

var title = d3.select("svg").append("g")
    .attr("transform", "translate(" +margin.left+ "," +margin.top+ ")")
    .attr("class","title");

title.append("text")
    .attr("x", (w / 2))
    .attr("y", -30 )
    .attr("text-anchor", "middle")
    .style("font-size", "22px")
    .text("A donut chart");
</script>
</body>
</html>
```

For an example of multiseries data, you will add another column of data representing the expense to the file `data_04.csv` as shown in Listing 22-19, and you will save this new version as `data_06.csv`.

Listing 22-19. data_06.csv

```
country,income,expense
France,14,10
Russia,22,19
Japan,13,6
South Korea,34,12
Argentina,28,26
```

You have added a new set of data. Differently from the previous example, you must, therefore, create a new arc for this series. Then, in addition to the second arc, you add a third one. This arc is not going to draw the slices of a series, but you are going to use it in order to distribute labels circularly. These labels show the names of the countries, providing an alternative to a legend. Thus, divide the radius into three parts, leaving a gap in between to separate the series as shown in Listing 22-20.

Listing 22-20. ch22_02.html

```
var arc1 = d3.svg.arc()
    .outerRadius(0.4 * radius)
    .innerRadius(0.2 * radius);
var arc2 = d3.svg.arc()
    .outerRadius(0.7 * radius )
    .innerRadius(0.5 * radius );
```

```
var arc3 = d3.svg.arc()
    .outerRadius(radius)
    .innerRadius(0.8 * radius);
```

You have just created two arcs to manage the two series, and consequently it is now necessary to create two pies, one for the values of income and the other for the values of expenses (see Listing 22-21).

Listing 22-21. ch22_02.html

```
var pie = d3.layout.pie()
    .sort(null)
    .value(function(d) { return d.income; });
var pie2 = d3.layout.pie()
    .sort(null)
    .value(function(d) { return d.expense; });
```

You use the `d3.csv()` function to read the data within the file as shown in Listing 22-22. You do the usual iteration of data with `forEach()` for the interpretation of income and expense as numerical values.

Listing 22-22. ch22_02.html

```
d3.csv("data_06.csv", function(data) {

    data.forEach(function(d) {
       d.income = +d.income;
       d.expense = +d.expense;
    });

});
```

In Listing 22-23, you create the `path` elements which draw the various sectors of the two donuts, corresponding to the two series. With the functions `data()`, you bind the data of the two pie layouts to the two representations. Both the donuts must follow the same sequence of colors. Once the path element is defined, you connect it with a `text` element in which the corresponding numeric value is reported. Thus, you have added some labels which make it easier to read the chart.

Listing 22-23. ch22_02.html

```
var g = svg.selectAll(".arc1")
    .data(pie(data))
    .enter().append("g")
    .attr("class", "arc1");

g.append("path")
    .attr("d", arc1)
    .style("fill", function(d) { return color(d.data.country); });

g.append("text")
    .attr("transform", function(d) {
       return "translate(" + arc1.centroid(d) + ")"; })
    .attr("dy", ".35em")
    .style("text-anchor", "middle")
    .text(function(d) { return d.data.income; });
```

```
var g = svg.selectAll(".arc2")
    .data(pie2(data))
    .enter().append("g")
    .attr("class", "arc2");

g.append("path")
    .attr("d", arc2)
    .style("fill", function(d) { return color(d.data.country); });

g.append("text")
    .attr("transform", function(d) {
       return "translate(" + arc2.centroid(d) + ")"; })
    .attr("dy", ".35em")
    .style("text-anchor", "middle")
    .text(function(d) { return d.data.expense; });
```

Now, all that remains to do is to add the external labels carrying out the functions of the legend as shown in Listing 22-24.

Listing 22-24. ch22_02.html

```
g.append("text")
    .attr("transform", function(d) {
       return "translate(" + arc3.centroid(d) + ")"; })
    .style("text-anchor", "middle")
    .text(function(d) { return d.data.country; });
```

And so you get the donut chart shown in Figure 22-10.

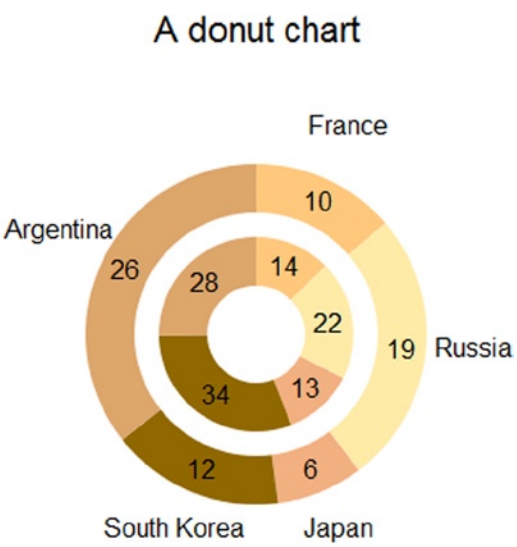


Figure 22-10. *A donut chart*

Polar Area Diagrams

Polar area diagrams are very similar to pie charts, but they differ in how far each sector extends from the center of the circle, giving the possibility to represent a further value. The extent of every slice is proportional to this new added value (see Figure 22-11).

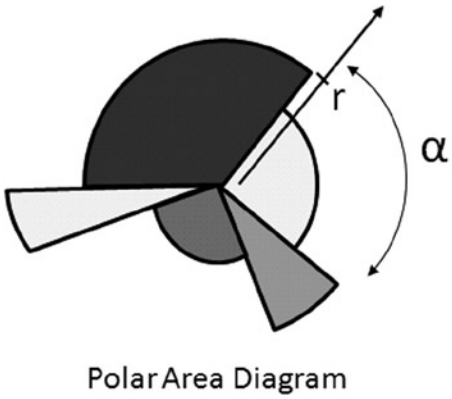


Figure 22-11. *In a polar area diagram, each slice is characterized by a radius r and an angle*

Consider the data in the file `data_04.csv` again and add an additional column which shows the growth of the corresponding country as shown in Listing 22-25. Save it as `data_07.csv`.

Listing 22-25. data_07.csvl

```
country,income,growth
France,14,10
Russia,22,19
Japan,13,9
South Korea,34,12
Argentina,28,16
```

Start writing the code in Listing 22-26; again, we will not explain this part because it is identical to the previous examples.

Listing 22-26. ch22_03.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<script src="http://d3js.org/d3.v3.js"></script>
<style>
body {
    font: 16px sans-serif;
}
```

```
.arc path {
    stroke: #000;
}
</style>
</head>
<body>
<script type="text/javascript">
var margin = {top: 70, right: 20, bottom: 30, left: 40},
    w = 500 - margin.left - margin.right,
    h = 400 - margin.top - margin.bottom;

var color = d3.scale.ordinal()
    .range(["#ffc87c", "#ffeba8", "#f3b080", "#916800", "#dda66b"]);

var radius = Math.min(w, h) / 2;

var pie = d3.layout.pie()
    .sort(null)
    .value(function(d) { return d.income; });

var svg = d3.select("body").append("svg")
    .attr("width", w + margin.left + margin.right)
    .attr("height", h + margin.top + margin.bottom)
    .append("g")
    .attr("transform", "translate(" +(w/2-margin.left)+
       "," +(h/2+margin.top)+ ")");

var title = d3.select("svg").append("g")
    .attr("transform", "translate(" +margin.left+ "," +margin.top+ ")")
    .attr("class","title");

title.append("text")
    .attr("x", (w / 2))
    .attr("y", -30 )
    .attr("text-anchor", "middle")
    .style("font-size", "22px")
    .text("A polar area diagram");
</script>
</body>
</html>
```

In Listing 22-27, you read the data in the `data_07.csv` file with the `d3.csv()` function and make sure that the values of income and growth are interpreted as numeric values.

Listing 22-27. ch22_03.html

```
d3.csv("data_07.csv", function(error, data) {

    data.forEach(function(d) {
       d.income = +d.income;
       d.growth = +d.growth;
    });
});
```

Differently to the previous examples, here you define not only an arc, but an arc which will vary with the variation of data being read; we will call it `arcs`, since the `outerRadius` is no longer constant but is proportional to the growth values in the file. In order to do this, you need to apply a general iterative function, and then the arcs must be declared within the `d3.csv()` function (see Listing 22-28).

Listing 22-28. ch22_03.html

```
d3.csv("data_07.csv", function(error, data) {

    data.forEach(function(d) {
       d.income = +d.income;
       d.growth = +d.growth;
    });

    arcs = d3.svg.arc()
       .innerRadius( 0 )
       .outerRadius( function(d,i) { return 8*d.data.growth; });
});
```

Now, you just have to add the SVG elements which draw the slices with labels containing the values of growth and income (see Listing 22-29). The labels reporting income values will be drawn inside the slices, right at the value returned by the `centroid()` function. Instead, as regards the labels reporting the growth values, these will be drawn just outside the slices. To obtain this effect, you can use the x and y values returned by `centroid()` and multiply them by a value greater than 2. You must recall that the centroid is at the very center of the angle and in the middle between `innerRadius` and `outerRadius`. Therefore, multiplying them by 2, you get the point at the center of the outer edge of the slice. If you multiply them by a value greater than 2, then you will find x and y positions outside the slice, right where you want to draw the label with the value of growth.

Listing 22-29. ch22_03.html

```
var g = svg.selectAll(".arc")
    .data(pie(data))
    .enter().append("g")
    .attr("class", "arc");

g.append("path")
    .attr("d", arcs)
    .style("fill", function(d) { return color(d.data.country); });

g.append("text")
    .attr("class","growth")
    .attr("transform", function(d) {
    a = arcs.centroid(d)[0]*2.2;
       b = arcs.centroid(d)[1]*2.2;
       return "translate(" +a+","+b+ ")"; })
    .attr("dy", ".35em")
    .style("text-anchor", "middle")
    .text(function(d) { return d.data.growth; });

g.append("text")
    .attr("class","income")
    .attr("transform", function(d) {
```

```
        return "translate(" +arcs.centroid(d)+ ")"; })
    .attr("dy", ".35em")
    .style("text-anchor", "middle")
    .text(function(d) { return d.data.income; });
```

One thing you have not yet done to the pie chart is adding a legend. In Listing 22-30, we define, outside the `d3.csv()` function, an element `<g>` in which to insert the table of the legend, and inside the function we define all the elements related to the countries, since defining them requires access to the values written in the file.

Listing 22-30. ch22_03.html

```
var legendTable = d3.select("svg").append("g")
    .attr("transform", "translate(" +margin.left+ ","+margin.top+")")
    .attr("class","legendTable");

d3.csv("data_07.csv", function(error, data) {
...
    var legend = legendTable.selectAll(".legend")
      .data(pie(data))
      .enter().append("g")
      .attr("class", "legend")
      .attr("transform", function(d, i) {
         return "translate(0," + i * 20 + ")"; });

    legend.append("rect")
       .attr("x", w - 18)
       .attr("y", 4)
       .attr("width", 10)
       .attr("height", 10)
       .style("fill", function(d) { return color(d.data.country); });

    legend.append("text")
       .attr("x", w - 24)
       .attr("y", 9)
       .attr("dy", ".35em")
       .style("text-anchor", "end")
       .text(function(d) { return d.data.country; });
});
```

Finally, you can make some adjustments to the CSS style classes as shown in Listing 22-31.

Listing 22-31. ch22_03.html

```
<style>
body {
    font: 16px sans-serif;
}
.arc path {
    stroke: #fff;
    stroke-width: 4;
}
```

```
.arc .income {
    font: 12px Arial;
    color: #fff;
}
</style>
```

And here is the polar area diagram (see Figure 22-12).

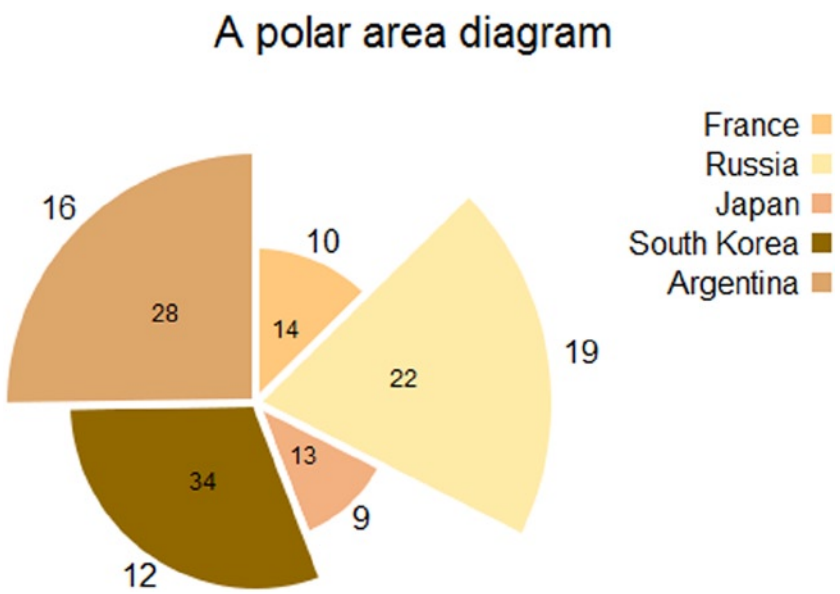


Figure 22-12. *A polar area diagram*

Summary

In this chapter, you learned how to implement pie charts and donut charts using the D3 library, following almost the same guidelines as those provided in the previous chapters. Furthermore, at the end of the chapter you learned how to make a polar area diagram, a type of chart which you had not met before and which the D3 library allows you to implement easily.

In the next chapter, you will implement the two types of Candlestick chart which you had already discussed in the first part of the book covering the jqPlot library, only this time you will be using the D3 library.

CHAPTER 23

Candlestick Charts with D3

In this short but nonetheless important chapter, you will look at candlestick charts. This type of chart is based on a particular data format (OHLC, or open-high-low-close) that you have already dealt with when the jqPlot library was covered (see Chapter 12). With jqPlot, you had a special plug-in to deal with and represent such data in an appropriate manner; instead, with D3 you have to build all of the graphic elements one by one, and above all you will need to implement a parser to read OHLC data from an external file. Moreover, another nontrivial aspect that you need to solve is how to deal with date and time data.

Although this sounds complex, in this chapter you will discover how the D3 library provides you with tools which make things easy and immediate for you.

You will first begin with building a simple OHLC chart, in order to focus particularly on the reading of the OHLC data. Then you will look in detail at how D3 handles date and time data, and finally you will represent the OHLC chart using only scalar vector graphics (SVG) elements such as lines.

In the last part, you will convert your OHLC chart in a more complete candlestick chart by means of only a few modifications.

Creating an OHLC Chart

Because of the capability of D3 to build new graphical structures from small graphical components, you can also create candlestick charts such as those generated with jqPlot. You have already seen that a candlestick chart requires a well-defined data structure: a timeline of data which consists of a date and the four OHLC values. You copy the data from Listing 23-1 into a file and save it as `data_08.csv`.

Listing 23-1. data_08.csv

```
date,open,min,max,close,
08/08/2012,1.238485,1.2327,1.240245,1.2372,
08/09/2012,1.23721,1.22671,1.23873,1.229295,
08/10/2012,1.2293,1.22417,1.23168,1.228975,
08/12/2012,1.229075,1.22747,1.22921,1.22747,
08/13/2012,1.227505,1.22608,1.23737,1.23262,
08/14/2012,1.23262,1.23167,1.238555,1.232385,
08/15/2012,1.232385,1.22641,1.234355,1.228865,
08/16/2012,1.22887,1.225625,1.237305,1.23573,
08/17/2012,1.23574,1.22891,1.23824,1.2333,
08/19/2012,1.23522,1.23291,1.235275,1.23323,
08/20/2012,1.233215,1.22954,1.236885,1.2351,
08/21/2012,1.23513,1.23465,1.248785,1.247655,
08/22/2012,1.247655,1.24315,1.254415,1.25338,
```

```
08/23/2012,1.25339,1.252465,1.258965,1.255995,
08/24/2012,1.255995,1.248175,1.256665,1.2512,
08/26/2012,1.25133,1.25042,1.252415,1.25054,
08/27/2012,1.25058,1.249025,1.25356,1.25012,
08/28/2012,1.250115,1.24656,1.257695,1.2571,
08/29/2012,1.25709,1.251895,1.25736,1.253065,
08/30/2012,1.253075,1.248785,1.25639,1.25097,
08/31/2012,1.25096,1.249375,1.263785,1.25795,
09/02/2012,1.257195,1.256845,1.258705,1.257355,
09/03/2012,1.25734,1.25604,1.261095,1.258635,
09/04/2012,1.25865,1.25264,1.262795,1.25339,
09/05/2012,1.2534,1.250195,1.26245,1.26005,
09/06/2012,1.26006,1.256165,1.26513,1.26309,
09/07/2012,1.26309,1.262655,1.281765,1.281625,
09/09/2012,1.28096,1.27915,1.281295,1.279565,
09/10/2012,1.27957,1.27552,1.28036,1.27617,
09/11/2012,1.27617,1.2759,1.28712,1.28515,
09/12/2012,1.28516,1.281625,1.29368,1.290235,
```

In what by now has become almost a habit, you begin by writing the code which is common to almost all charts and does not require further explanations (see Listing 23-2).

Listing 23-2. ch23_01.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<script src="http://d3js.org/d3.v3.js"></script>
<style>
body {
    font: 16px sans-serif;
}
</style>
</head>
<body>
<script type="text/javascript">
var margin = {top: 70, right: 20, bottom: 30, left: 40},
    w = 500 - margin.left - margin.right,
    h = 400 - margin.top - margin.bottom;

var svg = d3.select("body").append("svg")
    .attr("width", w + margin.left + margin.right)
    .attr("height", h + margin.top + margin.bottom)
    .append("g")
    .attr("transform", "translate(" +margin.left+ "," +margin.top+ ")");

var title = d3.select("svg").append("g")
    .attr("transform", "translate(" +margin.left+ "," +margin.top+ ")")
    .attr("class","title");
```

```
title.append("text")
    .attr("x", (w / 2))
    .attr("y", -30 )
    .attr("text-anchor", "middle")
    .style("font-size", "22px")
    .text("My candlestick chart");
</script>
</body>
</html>
```

Since in the first column of the file there are values of date type, you need to define a parser to set their format (see Listing 23-3).

Listing 23-3. ch23_01.html

```
...
    w = 500 - margin.left - margin.right,
    h = 400 - margin.top - margin.bottom;

var parseDate = d3.time.format("%m/%d/%Y").parse;
...
```

A candlestick chart is a type of data representation which is generally temporal, i.e., the four OHLC data are related to a single time unit and their variations over time are visible along the x axis. You will therefore have an x axis on which you will have to handle time values, whereas on the y axis you will assign a linear scale. In defining the x axis, you make sure that the dates are reported showing only day and month, which will be indicated by the first three initial characters (see Listing 23-4).

Listing 23-4. ch23_01.html

```
var parseDate = d3.time.format("%m/%d/%Y").parse;

var x = d3.time.scale()
    .range([0, w]);

var y = d3.scale.linear()
    .range([h, 0]);

var xAxis = d3.svg.axis()
    .scale(x)
    .orient("bottom")
    .tickFormat(d3.time.format("%d-%b"))
    .ticks(5);

var yAxis = d3.svg.axis()
    .scale(y)
    .orient("left");

...
```

Now observing what is inside the data file (Listing 23-1), you can see five columns of data, of which the last four are numbers. The first column contains dates which must be submitted to the parser, while the other four are to be interpreted as numeric values. Moreover, you need to figure out which are the maximum and minimum values among all of the OHLC data. Manage all these aspects within the iterative function `forEach()` as shown in Listing 23-5.

Listing 23-5. ch23_01.html

```
...
var svg = d3.select("body").append("svg")
    .attr("width", w + margin.left + margin.right)
    .attr("height", h + margin.top + margin.bottom)
    .append("g")
    .attr("transform", "translate(" +margin.left+ "," +margin.top+ ")");

d3.csv("data_08.csv", function(error, data) {
    var maxVal = -1000;
    var minVal = 1000;
    data.forEach(function(d) {
       d.date = parseDate(d.date);
       d.open = +d.open;
       d.close = +d.close;
       d.max = +d.max;
       d.min = +d.min;
       if (d.max > maxVal)
          maxVal = d.max;
       if (d.min < minVal)
          minVal = d.min;
    });
});
...
```

Next, in Listing 23-6, you create the domains of x and y. While on the x axis, the domain will handle dates, the y domain will have an extension which will cover all the values between the minimum and maximum values that have just been found (`minVal` and `maxVal`).

Listing 23-6. ch23_01.html

```
d3.csv("data_08.csv", function(error, data) {

    data.forEach(function(d) {
    ...
    });

    x.domain(d3.extent(data, function(d) { return d.date; }));
    y.domain([minVal,maxVal]);
});
```

Once the domains are well defined you can draw the two axes x and y with the SVG elements along with their labels as shown in Listing 23-7.

Listing 23-7. ch23_01.html

```
d3.csv("data_08.csv", function(error, data) {
    ...
    y.domain([minVal,maxVal]);

    svg.append("g")
       .attr("class", "x axis")
       .attr("transform", "translate(0," + h + ")")
       .call(xAxis)

    svg.append("text")
       .attr("class", "label")
       .attr("x", w)
       .attr("y", -6)
       .style("text-anchor", "end");

    svg.append("g")
       .attr("class", "y axis")
       .call(yAxis);

    svg.append("text")
       .attr("class", "label")
       .attr("transform", "rotate(-90)")
       .attr("y", 6)
       .attr("dy", ".71em")
       .style("text-anchor", "end")
       .text("Dollar [$]");
});
```

And you use the SVG element `<line>` to plot the data on the OHLC chart (see Listing 23-8). The `ext` line is the vertical line which defines the range between the *high* and *low* values. The `close` and `open` lines are two horizontal lines corresponding to *open* and *close* values.

Listing 23-8. ch23_01.html

```
d3.csv("data_08.csv", function(error, data) {
    ...
    svg.append("text")
      .attr("class", "label")
      .attr("transform", "rotate(-90)")
      .attr("y", 6)
      .attr("dy", ".71em")
      .style("text-anchor", "end")
      .text("Dollar [$]")

    svg.selectAll("line.ext")
       .data(data)
       .enter().append("svg:line")
       .attr("class", "ext")
       .attr("x1", function(d) { return x(d.date)})
```

```
        .attr("x2", function(d) { return x(d.date)})
        .attr("y1", function(d) { return y(d.min);})
        .attr("y2", function(d) { return y(d.max); });

    svg.selectAll("line.close")
        .data(data)
        .enter().append("svg:line")
        .attr("class", "close")
        .attr("x1", function(d) { return x(d.date)+5})
        .attr("x2", function(d) { return x(d.date)-1})
        .attr("y1", function(d) { return y(d.close);})
        .attr("y2", function(d) { return y(d.close); });

    svg.selectAll("line.open")
        .data(data)
        .enter().append("svg:line")
        .attr("class", "open")
        .attr("x1", function(d) { return x(d.date)+1})
        .attr("x2", function(d) { return x(d.date)-5})
        .attr("y1", function(d) { return y(d.open);})
        .attr("y2", function(d) { return y(d.open); });
});
```

Thanks to the way in which you have defined the classes of the newly generated elements, you can define attributes to the CSS styles for all of the three lines together, by using the `line` class, or defining them individually using the `line.open`, `line.close`, and `line.ext` classes (see Listing 23-9).

Listing 23-9. ch23_01.html

```
<style>
body {
    font: 16px sans-serif;
}
.axis path,
.axis line {
    fill: none;
    stroke: #000;
    shape-rendering: crispEdges;
}
line.open, line.close, line.ext {
    stroke: blue;
    stroke-width: 2;
    shape-rendering: crispEdges;
}
</style>
```

At the end, you get the candlestick chart shown in Figure 23-1, which has nothing to envy those obtained with jqPlot.

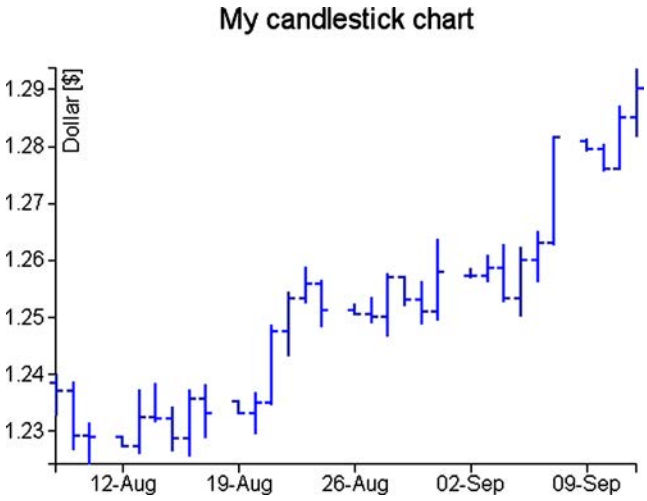


Figure 23-1. *An OHLC chart*

Date Format

Having to deal with this class of charts which makes use of OHLC data, you will always deal with time and date values along the x axis. Therefore taking a cue from this observation, analyze how the D3 library handles this type of data.

What would have happened if in the previous example you did not have the dates of the days and months zero-padded, or the year was reported with only two digits (e.g., "8/9/12")? Inside the `d3.csv()` function, D3 would not have been able to read dates with this format, and consequently, the candlestick chart would not have appeared. Actually, what you need to do is very simple, i.e., guess the correct sequence of formatters to insert in the parser. By formatter, we mean a set of characters with the "%" sign before, which according to the specific (case-sensitive) character expresses a unit of time written in a certain way.

```
var parseDate = d3.time.format("%m/%e/%y").parse;
```

Even dates expressed literally can be handled in the same way. You have already seen this format of dates:

```
08-Aug-12,1.238485,1.2327,1.240245,1.2372,
```

It can be handled with this parser:

```
var parseDate = d3.time.format("%d-%b-%y").parse;
```

But there are much more complex cases, such as the following:

```
Monday 16 April 2012,1.238485,1.2327,1.240245,1.2372,
```

And it can be handled with this parser:

```
var parseDate = d3.time.format("%A %e %B %Y").parse;
```

All separation characters (including spaces) between the different values should be reported in the same position in the parser. Thus, if the dates are defined in this way…

```
'8 Aug-12',1.238485,1.2327,1.240245,1.2372,
```

You have to insert both the space and the quotes in the string defining the parser or the date would not be recognized.

```
var parseDate = d3.time.format("'%d %b-%y'").parse;
```

You must also keep in mind that the only separation character which can not be added in a csv file is ", ". If you must insert it, you have to use a TSV (tab-separated values) file.

Table 23-1 includes all the available formatters. Their combination should cover any input size.

Table 23-1. *D3 Date and Time Formatters*

Formatter	Description
%a	Abbreviated weekday name
%A	Full weekday name
%b	Abbreviated month name
%B	Full month name
%c	Date and time, as "%a %b %e %H:%M:%S %Y"
%d	Zero-padded day of the month as a decimal number [01,31]
%e	Space-padded day of the month as a decimal number [1,31]
%H	Hour (24-hour clock) as a decimal number [00,23]
%I	Hour (12-hour clock) as a decimal number [01,12]
%j	Day of the year as a decimal number [001,366]
%m	Month as a decimal number [01,12]
%M	Minute as a decimal number [00,59]
%p	Either AM or PM
%S	Second as a decimal number [00,61]
%U	Week number of the year (Sunday as the first day of the week) as a decimal number [00,53]
%w	Weekday as a decimal number [0(Sunday),6]
%W	Week number of the year (Monday as the first day of the week) as a decimal number [00,53]
%x	Date, as "%m/%d/%y"
%X	Time, as "%H:%M:%S"
%y	Year without century as a decimal number [00,99]
%Y	Year with century as a decimal number
%Z	Time zone offset, such as "–0700"
%%	A literal "%" character

Box Representation in Candlestick Charts

With jqPlot, you also saw other ways to display OHLC data. For example, often such data are represented by a vertical line and a vertical box covering it for a certain length. The vertical line is the same representation as the previous candlestick, it lies between the *high* and *low* value of the OHLC. Instead, the box represents the range between the *open* and *close* values. Moreover, if the *open* value is greater than the *close* value, the box will be of a given color, but if the opposite happens, of another color.

You use the same data contained in the `data_08.csv` file, and starting from the code in the previous example, you will look at the changes to be made.

Replace the `ext`, `open`, and `close` lines with these three new lines: `ext`, `ext1`, and `ext2` (see Listing 23-10). Then you have to add the rectangle representing the box. The lines should be black, whereas the boxes should be red when the *open* values are greater than *close* values, or else, in the opposite case, the boxes will be green.

Listing 23-10. ch23_02.html

```
svg.selectAll("line.ext")
    .data(data)
    .enter().append("svg:line")
    .attr("class", "ext")
    .attr("x1", function(d) { return x(d.date)})
    .attr("x2", function(d) { return x(d.date)})
    .attr("y1", function(d) { return y(d.min);})
    .attr("y2", function(d) { return y(d.max);});

svg.selectAll("line.ext1")
    .data(data)
    .enter().append("svg:line")
    .attr("class", "ext")
    .attr("x1", function(d) { return x(d.date)+3})
    .attr("x2", function(d) { return x(d.date)-3})
    .attr("y1", function(d) { return y(d.min);})
    .attr("y2", function(d) { return y(d.min); });

svg.selectAll("line.ext2")
    .data(data)
    .enter().append("svg:line")
    .attr("class", "ext")
    .attr("x1", function(d) { return x(d.date)+3})
    .attr("x2", function(d) { return x(d.date)-3})
    .attr("y1", function(d) { return y(d.max);})
    .attr("y2", function(d) { return y(d.max); });

svg.selectAll("rect")
    .data(data)
    .enter().append("svg:rect")
    .attr("x", function(d) { return x(d.date)-3; })
    .attr("y", function(d) { return y(Math.max(d.open, d.close));})
    .attr("height", function(d) {
    return y(Math.min(d.open, d.close))-y(Math.max(d.open, d.close));})
    .attr("width",6)
    .attr("fill",function(d) {
    return d.open > d.close ? "darkred" : "darkgreen" ;});
});
```

The last thing is to set the CSS style classes in Listing 23-11.

Listing 23-11. ch23_02.html

```
<style>
body {
  font: 16px sans-serif;
}
.axis path,
.axis line {
    fill: none;
    stroke: #000;
    shape-rendering: crispEdges;
}
line.ext, line.ext1, line.ext2 {
    stroke: #000;
    stroke-width: 1;
    shape-rendering: crispEdges;
}
</style>
```

And the chart in Figure 23-2 is the result.

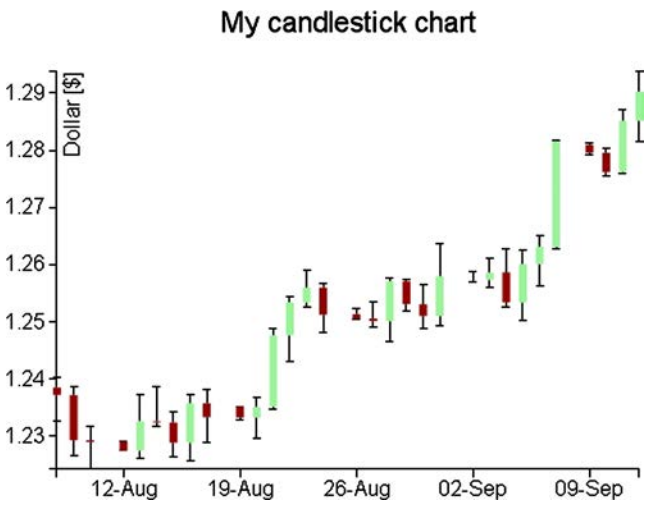


Figure 23-2. *A candlestick chart*

Summary

In this chapter, you have seen the types of **candlestick chart** already discussed in the first part of the book dedicated to the jqPlot library, but this time you used D3. You have seen how you can easily get similar results while keeping full control over every single graphic element. In addition, since this kind of chart uses time data, here you have delved deeper into how the D3 library manages this type of data and the various ways to manage format.

Continuing to follow the parallelism between the jqPlot library and the D3 library regarding the implementation of the various types of chart, in the next chapter you will learn about **scatter plots** and **bubble charts** and how to implement them with the D3 library.

CHAPTER 24

Scatterplot and Bubble Charts with D3

In this chapter, you will learn about scatterplot charts. Whenever you have a set of data pairs [x, y] and you want to analyze their distribution in the xy plane, you will refer to this type of chart. Thus, you will see first how to make this type of chart using the D3 library. In the first example, you will begin reading a TSV (tab-separated values) file containing more than one series of data, and through them, you will see how to achieve a scatterplot.

Once the scatterplot is completed, you will see how to represent the data points using markers with particular shapes, either choosing them from a predefined set or by creating original.

This class of charts is very important. It is a fundamental tool for analyzing data distributions; in fact, from these diagrams you can find particular trends (trendlines) and groupings (clusters). In this chapter, two simple examples will show you how you can represent trendlines and clusters.

Moreover, you will see how to add the highlighting functionality to your charts by the event handling and how the D3 library manages it.

Finally, the chapter will close with a final example in which you will need to represent data with three parameters [x, y, z]. Therefore, properly modifying the scatterplot, you will discover how you can get a bubble chart, which is a scatterplot modified to be able to represent an additional parameter.

Scatterplot

Thanks to the D3 library, there is no limit to the graphic representations which you can generate, combining graphical elements as if they were bricks. The creation of scatterplots is no exception.

You begin with a collection of data (see Listing 24-1), this time in a tabulated form (therefore a TSV file) which you will copy and save as a file named `data_09.tsv`. (See the following Note.)

Note Notice that the values in a TSV file are TAB separated, so when you write or copy Listing 24-1, remember to check that there is only a TAB character between each value.

Listing 24-1. data_09.tsv

```
time    intensity    group
10    171.11    Exp1
14    180.31    Exp1
17    178.32    Exp1
42    173.22    Exp3
30    145.22    Exp2
30    155.68    Exp3
```

```
23    200.56    Exp2
15    192.33    Exp1
24    173.22    Exp2
20    203.78    Exp2
18    187.88    Exp1
45    180.00    Exp3
27    181.33    Exp2
16    198.03    Exp1
47    179.11    Exp3
27    175.33    Exp2
28    162.55    Exp2
24    208.97    Exp1
23    200.47    Exp1
43    165.08    Exp3
27    168.77    Exp2
23    193.55    Exp2
19    188.04    Exp1
40    170.36    Exp3
21    184.98    Exp2
15    197.33    Exp1
50    188.45    Exp3
23    207.33    Exp1
28    158.60    Exp2
29    151.31    Exp2
26    172.01    Exp2
23    191.33    Exp1
25    226.11    Exp1
60    198.33    Exp3
```

Suppose that the data contained in the file belong to three different experiments (labeled as Exp1, Exp2, and Exp3), each applied to a different object (for example, three luminescent substances), in which you want to measure how their emission intensity varies over time. The readings are done repeatedly and at different times. Your aim will be to represent these values in the xy plane in order to analyze their distribution and eventual properties.

Observing the data, you can see that they are composed of three columns: time, intensity, and group membership. This is a typical data structure which can be displayed in the form of a scatterplot. You will put the time scale on the x axis, put the intensity values on the y axis, and finally identify groups by the shape or by the color of the markers which will mark the position of the point in the scatterplot.

As it has become customary, you begin by writing the code in Listing 24-2. This code represents your starting code, and since it is common to almost all charts you have seen in the previous example, it does not require further explanation.

Listing 24-2. ch24_01.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<script src="http://d3js.org/d3.v3.js"></script>
<style>
body {
    font: 16px sans-serif;
}
```

```
.axis path,
.axis line {
    fill: none;
    stroke: #000;
    shape-rendering: crispEdges;
}
</style>
</head>
<body>
<script type="text/javascript">
var margin = {top: 70, right: 20, bottom: 40, left: 40},
    w = 500 - margin.left - margin.right,
    h = 400 - margin.top - margin.bottom;

var color = d3.scale.category10();

var x = d3.scale.linear()
    .range([0, w]);

var y = d3.scale.linear()
    .range([h, 0]);

var xAxis = d3.svg.axis()
    .scale(x)
    .orient("bottom");

var yAxis = d3.svg.axis()
    .scale(y)
    .orient("left");

var svg = d3.select("body").append("svg")
    .attr("width", w + margin.left + margin.right)
    .attr("height", h + margin.top + margin.bottom)
    .append("g")
    .attr("transform", "translate(" + margin.left+ "," +margin.top+ ")");


var title = d3.select("svg").append("g")
    .attr("transform", "translate(" + margin.left+ "," +margin.top+ ")")
    .attr("class","title");

title.append("text")
    .attr("x", (w / 2))
    .attr("y", -30 )
    .attr("text-anchor", "middle")
    .style("font-size", "22px")
    .text("My Scatterplot");
</script>
</body>
</html>
```

In Listing 24-3, you read the tabulated data from the TSV file with the `d3.tsv()` function, making sure that the numerical values will be read as such. Here, even if you have times on the first column, these do not require parsing since they are seconds and can thus be considered on a linear scale.

Listing 24-3. ch24_01.html

```
...
var svg = d3.select("body").append("svg")
    .attr("width", w + margin.left + margin.right)
    .attr("height", h + margin.top + margin.bottom)
    .append("g")
    .attr("transform", "translate(" + margin.left+ "," +margin.top+ ")");

d3.tsv("data_09.tsv", function(error, data) {
    data.forEach(function(d) {
       d.time = +d.time;
       d.intensity = +d.intensity;
    });
});

var title = d3.select("svg").append("g")
    .attr("transform", "translate(" + margin.left+ "," +margin.top+ ")")
    .attr("class","title");
...
```

Also with regard to the domains, the assignment is very simple, as shown in Listing 24-4. Furthermore, you will use the `nice()` function, which rounds off the values of the domain.

Listing 24-4. ch24_01.html

```
d3.tsv("data_09.tsv", function(error, data) {
    data.forEach(function(d) {
       d.time = +d.time;
       d.intensity = +d.intensity;
    });

    x.domain(d3.extent(data, function(d) { return d.time; })).nice();
    y.domain(d3.extent(data, function(d) { return d.intensity; })).nice();
});
```

You add also the axis label, bringing “Time [s]” on the x axis and “Intensity” on the y axis, as shown in Listing 24-5.

Listing 24-5. ch24_01.html

```
d3.tsv("data_09.tsv", function(error, data) {
    ...
     x.domain(d3.extent(data, function(d) { return d.time; })).nice();
     y.domain(d3.extent(data, function(d) { return d.intensity; })).nice();

    svg.append("g")
       .attr("class", "x axis")
       .attr("transform", "translate(0," + h + ")")
       .call(xAxis);
```

```
    svg.append("text")
       .attr("class", "label")
       .attr("x", w)
       .attr("y", h + margin.bottom - 5)
       .style("text-anchor", "end")
       .text("Time [s]");

    svg.append("g")
       .attr("class", "y axis")
       .call(yAxis);

    svg.append("text")
       .attr("class", "label")
       .attr("transform", "rotate(-90)")
       .attr("y", 6)
       .attr("dy", ".71em")
       .style("text-anchor", "end")
       .text("Intensity");
});
```

Finally, you have to draw the markers directly on the graph. These can be represented by the SVG element `<circle>`. The data points to be represented on the scatterplot will therefore be of small dots of radius 3.5 pixels (see Listing 24-6). To define their representation of different groups, the markers are drawn in different colors.

Listing 24-6. ch24_01.html

```
d3.tsv("data_09.tsv", function(error, data) {
    ...
    svg.append("text")
      .attr("class", "label")
      .attr("transform", "rotate(-90)")
      .attr("y", 6)
      .attr("dy", ".71em")
      .style("text-anchor", "end")
      .text("Intensity");

    svg.selectAll(".dot")
       .data(data)
       .enter().append("circle")
       .attr("class", "dot")
       .attr("r", 3.5)
       .attr("cx", function(d) { return x(d.time); })
       .attr("cy", function(d) { return y(d.intensity); })
       .style("fill", function(d) { return color(d.group); });
});
```

Now you have so many colored markers on the scatterplot, but no reference to their color and the group to which they belong. Therefore, it is necessary to add a legend showing the names of the various groups associated with the different colors (see Listing 24-7).

Listing 24-7. ch24_01.html

```
d3.tsv("data_09.tsv", function(error, data) {
    ...
    svg.selectAll(".dot")
       .data(data)
       .enter().append("circle")
       .attr("class", "dot")
       .attr("r", 3.5)
       .attr("cx", function(d) { return x(d.time); })
       .attr("cy", function(d) { return y(d.intensity); })
       .style("fill", function(d) { return color(d.group); });

    var legend = svg.selectAll(".legend")
       .data(color.domain())
       .enter().append("g")
       .attr("class", "legend")
       .attr("transform", function(d, i) {
          return "translate(0," + (i * 20) + ")"; });

    legend.append("rect")
       .attr("x", w - 18)
       .attr("width", 18)
       .attr("height", 18)
       .style("fill", color);

    legend.append("text")
       .attr("x", w - 24)
       .attr("y", 9)
       .attr("dy", ".35em")
       .style("text-anchor", "end")
       .text(function(d) { return d; });
});
```

After all the work is complete, you get the scatterplot shown in Figure 24-1.

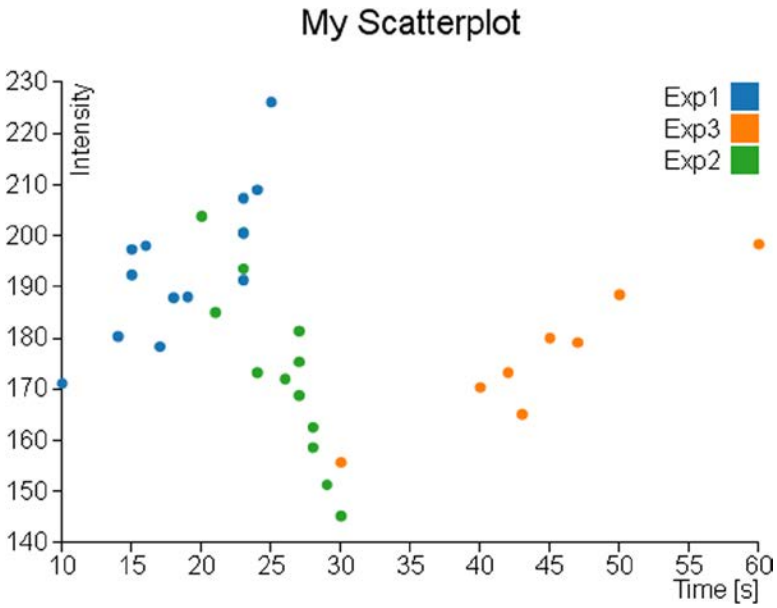


Figure 24-1. *A scatterplot showing the data distribution*

Markers and Symbols

When you want to represent a scatterplot, an aspect not to be underestimated is the shape of the marker with which you want to represent the data points. Not surprisingly, the D3 library provides you with a number of methods that manage the marker representation by symbols. In this chapter, you will learn about this topic since it is well suited to this kind of chart (scatterplots), but does not alter its application to other types of chart (e.g., line charts).

Using Symbols as Markers

D3 library provides a set of symbols that can be used directly as a marker. In Table 24-1, you can see a list reporting various predefined symbols.

Table 24-1. *Predefined Symbols in D3 Library*

Symbol	Description
Circle	A circle
Cross	A Greek cross (or plus sign)
Diamond	A rhombus
Square	An axis-aligned square
Triangle-down	A downward-pointing equilateral triangle
Triangle-up	An upward-pointing equilateral triangle

Continuing with the previous example, you replace the dots in the scatterplot chart with different symbols used as markers. These symbols will vary depending on the series of membership of the data (`Exp1`, `Exp2`, or `Exp3`). So this time, to characterize the series to which data belong, it will be both the color and the shape of the marker.

First, you need to assign each series to a symbol within the `groupMarker` object, as shown in Listing 24-8.

Listing 24-8. ch24_01b.html

```
var margin = {top: 70, right: 20, bottom: 40, left: 40},
    w = 500 - margin.left - margin.right,
    h = 400 - margin.top - margin.bottom;

var groupMarker = {
    Exp1: "cross",
    Exp2: "diamond",
    Exp3: "triangle-down"
};

var color = d3.scale.category10();
```

Then, you delete from the code the lines concerning the representation of the dots (see Listing 24-9). These lines will be replaced with others generating the markers (see Listing 24-10).

Listing 24-9. ch24_01b.html

```
svg.selectAll(".dot")
      .data(data)
      .enter().append("circle")
      .attr("class", "dot")
      .attr("r", 3.5)
      .attr("cx", function(d) { return x(d.time); })
      .attr("cy", function(d) { return y(d.intensity); })
      .style("fill", function(d) { return color(d.group); });
```

Actually, you are going to generate symbols that are nothing more than the predefined SVG paths. You may guess this from the fact that in Listing 24-10 the addition of the symbols is performed through the use of the `append("path")` function. Concerning instead the generation of the symbol as such, the D3 library provides a specific function: `d3.svg.symbol()`. The symbol to be displayed is passed as argument through the `type()` function, for example if you want to use the symbols to cross utilize `type("cross")`.

In this case, however, the symbols to be represented are three and they depend on the series of each point. So, you have to implement an iteration on all data by function `(d)` applied to `groupMarker`, which will return the string corresponding to the "cross", "diamond", and "triangle-down" symbols.

Finally, being constituted by a SVG path, the symbol can also be changed by adjusting the Cascading Style Sheets (CSS) styles. In this example, you can choose to represent only the outlines of the symbols by setting the `fill` attribute to `white`.

Listing 24-10. ch24_01b.html

```
d3.tsv("data_09.tsv", function(error, data) {
    ...
    svg.append("text")
      .attr("class", "label")
      .attr("transform", "rotate(-90)")
      .attr("y", 6)
```

```
      .attr("dy", ".71em")
      .style("text-anchor", "end")
      .text("Intensity");

   svg.selectAll("path")
      .data(data)
      .enter().append("path")
      .attr("transform", function(d) {
         return "translate(" + x(d.time) + "," + y(d.intensity) + ")";
      })
      .attr("d", d3.svg.symbol().type( function(d) {
         return groupMarker[d.group];
      }))
      .style("fill", "white")
      .style("stroke", function(d) { return color(d.group); })
      .style("stroke-width", "1.5px");

   var legend = svg.selectAll(".legend")
...
});
```

Figure 24-2 shows the scatterplot using various symbols in place of dots.

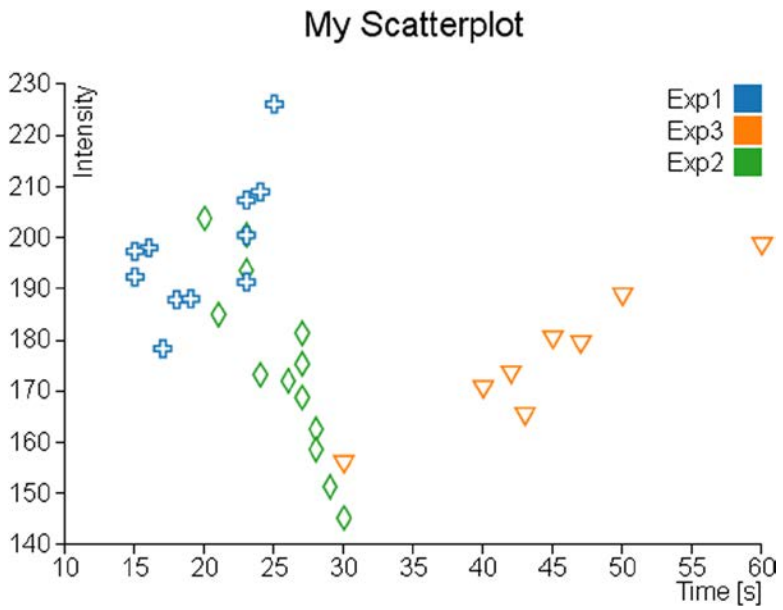


Figure 24-2. *In a scatterplot, the series could be represented by different symbols*

Using Customized Markers

You have just seen that the markers with the D3 library are nothing more than SVG paths. You could use this to your advantage by customizing your chart with the creation of other symbols that will be added to those already defined.

On the Internet, you can find a huge number of SVG symbols; once you have decided what symbol to use, you get its path in order to add it in your web page. More enterprising readers can also decide to edit SVG symbols with a SVG editor. I suggest you to use the Inkscape editor (see Figure 24-3); you can download it from its official site: `http://inkscape.org`. Or, more simply, you can start from an already designed SVG symbol and then modify it according to your taste. To do this, I recommend using the SVG Tryit page at this link: `www.w3schools.com/svg/tryit.asp?filename=trysvg_path` (see Figure 24-4).

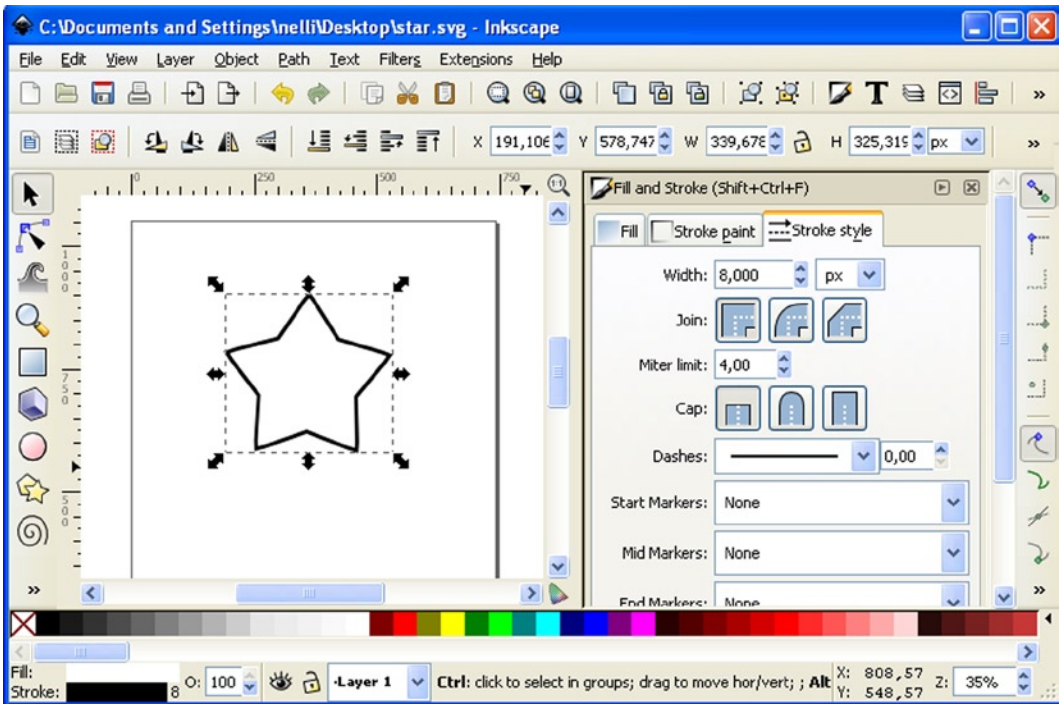


Figure 24-3. *Inkscape: a good SVG editor for generating symbols*

Figure 24-4. *Tryit is a useful tool to preview SVG symbols in real time inserting the path*

Therefore, choose three new symbols (e.g., a crescent moon, a star, and the Mars symbol) that go to replace the default ones. You extract their path and then insert into the definition of a new object, which you call `markers`, as shown in Listing 24-11.

Listing 24-11. ch24_01c.html

```
var margin = {top: 70, right: 20, bottom: 40, left: 40},
    w = 500 - margin.left - margin.right,
    h = 400 - margin.top - margin.bottom;

var markers = {
    mars: "m15,7 a 7,7 0 1,0 2,2 z l 1,1 7-7m-7,0 h 7 v 7",
    moon: "m15,3 a 8.5,8.5 0 1,0 0,13 a 6.5,6.5 0 0,1 0,-13",
    star: "m11,1 3,9h9l-7,5.5 2.5,8.5-7.5-5-7.5,5 2.5-8.5-7-6.5h9z"
};

var groupMarker = {
    ...
```

Now you have to update the associations between symbols and groups that you defined within the `groupMarker` variable, as shown in Listing 24-12.

Listing 24-12. ch24_01c.html

```
var groupMarker = {
    Exp1: markers.star,
    Exp2: markers.moon,
    Exp3: markers.mars
};
```

The last thing you can do is to change the definition of the path when you are creating the SVG elements (see Listing 24-13).

Listing 24-13. ch24_01c.html

```
svg.selectAll("path")
    .data(data)
    .enter().append("path")
    .attr("transform", function(d) {
       return "translate(" + x(d.time) + "," + y(d.intensity) + ")";
    })
    .attr("d", function(d) { return groupMarker[d.group]; })
    .style("fill", "white")
    .style("stroke", function(d) { return color(d.group); })
    .style("stroke-width", "1.5px");
```

At the end, you obtain a scatterplot reporting the symbols that you have personally created or downloaded from the Internet (see Figure 24-5).

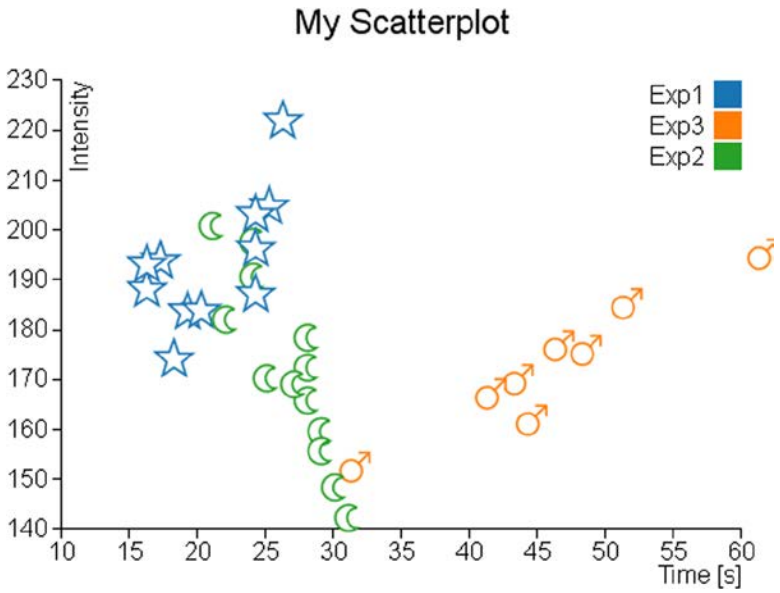


Figure 24-5. *A scatterplot with a customized set of markers*

Adding More Functionalities

Now that you have learned how to represent a distribution of data using scatterplots, it is time to introduce the trendline and clusters. Very often, analyzing in detail some sets of points in the data distribution, you can see that they follow a particular trend or tend to congregate in clusters. Therefore, it will be very useful to highlight this graphically. In this section, you will see a first example of how to calculate and represent linear trendlines. Then, you will see a second example which will illustrate a possibility of how to highlight some clusters present in the xy plane.

Trendlines

As for the jqPlot library, in which you had a plug-in that gave you the trendline directly, with the D3 library you need to implement not only the graphics but also its calculation.

For reasons of simplicity, you will calculate the trendline of a set of points (a series) following a linear trend. To do this, you use the method of *least squares*. This method ensures that you find, given a set of data, the line that best fits the trend of the points, as much as possible by minimizing the error (the sum of the squares of the errors).

■ **Note** For further information, I suggest you visit the Wolfram MathWorld article at `http://mathworld.wolfram.com/LeastSquaresFitting.html`.

For this example, you will continue working with the code of the scatterplot, but excluding all the changes made with the insertion of symbols. To avoid unnecessary mistakes and more repetition, Listing 24-14 shows the code you need to use as the starting point for this example.

Listing 24-14. ch24_02.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<script src="http://d3js.org/d3.v3.js"></script>
<style>
body {
    font: 16px sans-serif;
}
.axis path, .axis line {
    fill: none;
    stroke: #000;
    shape-rendering: crispEdges;
}
</style>
</head>
<body>
<script type="text/javascript">
var margin = {top: 70, right: 20, bottom: 40, left: 40},
    w = 500 - margin.left - margin.right,
    h = 400 - margin.top - margin.bottom;
var color = d3.scale.category10();
var x = d3.scale.linear()
    .range([0, w]);
var y = d3.scale.linear()
    .range([h, 0]);
var xAxis = d3.svg.axis()
    .scale(x)
    .orient("bottom");
var yAxis = d3.svg.axis()
    .scale(y)
    .orient("left");
var svg = d3.select("body").append("svg")
    .attr("width", w + margin.left + margin.right)
    .attr("height", h + margin.top + margin.bottom)
    .append("g")
    .attr("transform", "translate(" +margin.left+ "," +margin.top+ ")");
d3.tsv("data_09.tsv", function(error, data) {
    data.forEach(function(d) {
       d.time = +d.time;
       d.intensity = +d.intensity;
    });
    x.domain(d3.extent(data, function(d) { return d.time; })).nice();
    y.domain(d3.extent(data, function(d) { return d.intensity; })).nice();
    svg.append("g")
       .attr("class", "x axis")
       .attr("transform", "translate(0," + h + ")")
       .call(xAxis);
```

```
    svg.append("text")
       .attr("class", "label")
       .attr("x", w)
       .attr("y", h + margin.bottom - 5)
       .style("text-anchor", "end")
       .text("Time [s]");
    svg.append("g")
       .attr("class", "y axis")
       .call(yAxis);
    svg.append("text")
       .attr("class", "label")
       .attr("transform", "rotate(-90)")
       .attr("y", 6)
       .attr("dy", ".71em")
       .style("text-anchor", "end")
       .text("Intensity");
    svg.selectAll(".dot")
       .data(data)
       .enter().append("circle")
       .attr("class", "dot")
       .attr("r", 3.5)
       .attr("cx", function(d) { return x(d.time); })
       .attr("cy", function(d) { return y(d.intensity); })
       .style("fill", function(d) { return color(d.group); });
    var legend = svg.selectAll(".legend")
       .data(color.domain())
       .enter().append("g")
       .attr("class", "legend")
       .attr("transform", function(d, i) {
          return "translate(0," + (i * 20) + ")";
       });
    legend.append("rect")
       .attr("x", w - 18)
       .attr("width", 18)
       .attr("height", 18)
       .style("fill", color);
    legend.append("text")
       .attr("x", w - 24)
       .attr("y", 9)
       .attr("dy", ".35em")
       .style("text-anchor", "end")
       .text(function(d) { return d; });
});
var title = d3.select("svg").append("g")
    .attr("transform", "translate(" + margin.left + "," + margin.top + ")")
    .attr("class","title");
title.append("text")
    .attr("x", (w / 2))
    .attr("y", -30)
    .attr("text-anchor", "middle")
```

```
    .style("font-size", "22px")
    .text("My Scatterplot");
</script>
</body></html>
```

First, you define all the variables that will serve for the method of least squares within the `tsv()` function, as shown in Listing 24-15. For each variable you define an array of size 3, since there are three series to be represented in your chart.

Listing 24-15. ch24_02.html

```
d3.tsv("data_09.tsv", function(error, data) {

    sumx = [0,0,0];
    sumy = [0,0,0];
    sumxy = [0,0,0];
    sumx2 = [0,0,0];
    n = [0,0,0];
    a = [0,0,0];
    b = [0,0,0];
    y1 = [0,0,0];
    y2 = [0,0,0];
    x1 = [9999,9999,9999];
    x2 = [0,0,0];
    colors = ["","",""];

    data.forEach(function(d) {
    ...
});
```

Now you exploit the iteration of data performed during the parsing of data, to calculate simultaneously all the summations necessary for the method of least squares (see Listing 24-16). Moreover, it is convenient for the representation of a straight line to determine the maximum and minimum x values in each series.

Listing 24-16. ch24_02.html

```
d3.tsv("data_09.tsv", function(error, data) {
    ...
    data.forEach(function(d) {
       d.time = +d.time;
       d.intensity = +d.intensity;
       for(var i = 0; i < 3; i=i+1)
       {
          if(d.group == "Exp"+(i+1)){
             colors[i] = color(d.group);
             sumx[i] = sumx[i] + d.time;
             sumy[i] = sumy[i] + d.intensity;
             sumxy[i] = sumxy[i] + (d.time * d.intensity);
             sumx2[i] = sumx2[i] + (d.time * d.time);
             n[i] = n[i] +1;
```

```
            if(d.time < x1[i])
              x1[i] = d.time;
            if(d.time > x2[i])
              x2[i] = d.time;
          }
      }
    });

    x.domain(d3.extent(data, function(d) { return d.time; })).nice();
    ...
});
```

Once you have calculated all the summations, it is time to make the calculation of the least squares in Listing 24-17. Since the series are three, you will repeat the calculation for three times within a `for()` loop. Furthermore, within each loop you directly insert the creation of the SVG element for drawing the line corresponding to the result of each calculation.

Listing 24-17. ch24_02.html

```
d3.tsv("data_09.tsv", function(error, data) {
    ...
    x.domain(d3.extent(data, function(d) { return d.time; })).nice();
    y.domain(d3.extent(data, function(d) { return d.intensity; })).nice();

    for(var i = 0; i < 3; i = i + 1){
       b[i] = (sumxy[i] - sumx[i] * sumy[i] / n[i]) /
           (sumx2[i] - sumx[i] * sumx[i] / n[i]);
       a[i] = sumy[i] / n[i] - b[i] * sumx[i] / n[i];
       y1[i] = b[i] * x1[i] + a[i];
       y2[i] = b[i] * x2[i] + a[i];
       svg.append("svg:line")
          .attr("class","trendline")
          .attr("x1", x(x1[i]))
          .attr("y1", y(y1[i]))
          .attr("x2", x(x2[i]))
          .attr("y2", y(y2[i]))
          .style("stroke", colors[i])
          .style("stroke-width", 4);
    }
```

Now that you have completed the whole, you can see the representation of the three trendlines within the scatterplot, as shown in Figure 24-6.

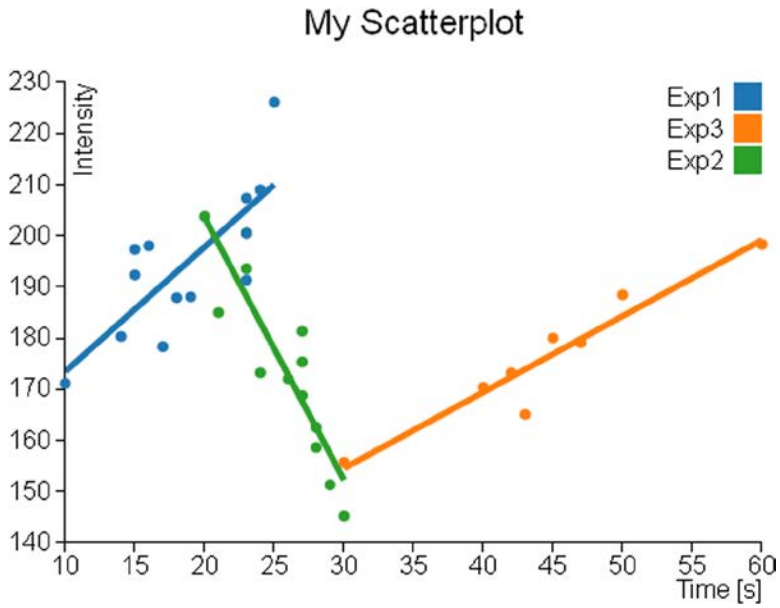


Figure 24-6. *Each group shows its trendline*

Clusters

When you work with the scatterplot, you may need to perform a clustering analysis. On the Internet, there are many analysis methods and algorithms that allow you to perform various operations of identification and research of clusters.

The cluster analysis is a classification technique that has the aim to identify data groups (clusters precisely) within a distribution of data (in this case, the scatterplot on the xy plane). The assignment of the various data points to these clusters is not defined a priori, but it is the task of cluster analysis to determine the criteria for selection and grouping. These clusters should be differentiated as much as possible, and in this case, as the grouping criterion, the cluster analysis will be based on the exaxt distances between the various data points and a point representative of the cluster called centroid (see Figure 24-7).

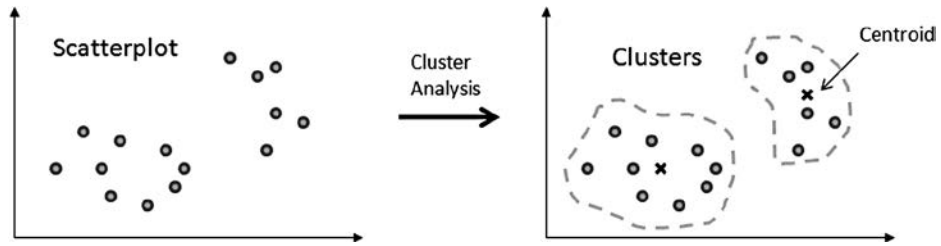


Figure 24-7. *The cluster analysis groups a set of data points around a centroid for each cluster*

Thus, the aim of this analysis is primarily to identify possible similarities within a data distribution, and in this regard there is nothing more appropriate of a scatterplot chart in which you can highlight these similarities through the different colors of the different points depending on the cluster membership.

In this section, you will see how to implement a cluster analysis algorithm and then how it is possible to integrate it into a scatterplot.

K-Mean Algorithm

Given the complexity of the cluster analysis, this chapter will not go into this topic in detail. You are interested only in highlighting the various points of the scatterplot in a different way from that of the series to which they belong (Exp1, Exp2, and Exp3). In this example, you want to color the data points depending on the cluster to which they belong. In order to do this, you will use a simple case of cluster analysis: the K-means algorithm. You define first the number of clusters into which you want to divide all the data, and then for each cluster, a representative point (centroid) is chosen. The distance between each data point and the three centroids is considered as a criterion for membership.

There are some examples available on the Internet in which the K-means method is applied, and it is totally implemented in JavaScript; among them I choose one developed by Heather Arthur (https://github.com/harthur/clusterfck), but you can replace it with any other.

For the example in question, I have taken the liberty to modify the code to make it as easy as possible. Starting from the data points contained within the TSV file, and representing them in a scatterplot, you are practically analyzing how these points are distributed in space xy. Now you are interested to recognize in this distribution, for example, three different clusters.

To do so you will apply the following algorithm:

1. Make a random choice of three data points as cluster centroids.
2. Iterate over all the data points in the file, assigning each of them to the cluster that has the closest centroid. At the end, you have all the data points divided into three clusters.
3. Within each cluster, a new centroid is calculated, which this time will not correspond to any given point but will be the "midpoint" interposed between all points in the cluster.
4. Recalculate steps 2 and 3 until the new centroids correspond to the previous ones (that is, the coordinates of the centroids in the xy plane remain unchanged).

Once the algorithm is completed, you will have the points in the scatterplot with three different colors corresponding to three different clusters.

Please note that in this algorithm there is no optimization, and thus the result will always be different every time you upload the page in your browser. In fact, what you get every time is a possible solution, not the "best solution."

Now to keep some of modularity, you will write the code of the analysis of clusters in an external file which you will call `kmeans.js`.

First, you will implement the `randomCentroids()` function, which will choose k points (in this example, k = 3) among those contained in the file (here passed within the `points` array) to assign them as centroids of the k clusters (see Listing 24-18). This function corresponds to the point 1 of the algorithm.

Listing 24-18. kmeans.js

```
function randomCentroids(points, k) {
    var centroids = points.slice(0);
    centroids.sort(function() {
       return (Math.round(Math.random()) - 0.5);
    });
    return centroids.slice(0, k);
};
```

Now you have to assign all the points contained in the file to the three different clusters. To do this, you need to calculate the distance between each data point and the centroid in question, and thus you need to implement

a specific function that calculates the distance between two points. In Listing 24-19, it is defined the `distance()` function, which returns the distance between `v1` and `v2` generic points.

Listing 24-19. kmeans.js

```
function distance(v1, v2) {
      var total = 0;
      for (var i = 0; i < v1.length; i++) {
         total += Math.pow((v2[i] - v1[i]), 2);
      }
      return Math.sqrt(total);
};
```

Now that you know how to calculate the distance between two points, you can implement a function that is able to decide which is the cluster assignment of each data point, calculating its distance with all centroids and choosing the smaller one. Thus, you can add the `closestCentroid()` function to the code, as shown in Listing 24-20.

Listing 24-20. kmeans.js

```
function closestCentroid(point, centroids) {
    var min = Infinity;
    var index = 0;
    for (var i = 0; i < centroids.length; i++) {
       var dist = distance(point, centroids[i]);
       if (dist < min) {
          min = dist;
          index = i;
       }
    }
    return index;
}:
```

Now you can write the function that expresses the algorithm first exposed in its entirety. This function requires two arguments, the input data points (`points`) and the number of clusters into which they will be divided (`k`) (see Listing 24-21). Within it, you then choose the centroids using the newly implemented `randomCentroids()` function (point 1 of the algorithm).

Listing 24-21. kmeans.js

```
function kmeans(points, k) {
    var centroids = randomCentroids(points, k);
};
```

Once you have chosen the three centroids, you can assign all data points (contained in the `points` array) to the three clusters, defining the `assignment` array as shown in Listing 24-22 (point 2 of the algorithm). This array has the same length of the points array and is constructed in such a way that the order of its elements corresponds to the order of data points. Every element contains the number of the cluster to which they belong. If, for example, in the third element of the `assignment` array you have a value of 2, then it will mean that the third data point belongs to the third cluster (clusters are 0, 1, and 2).

Listing 24-22. kmeans.js

```
function kmeans(points, k) {
    var centroids = randomCentroids(points, k);
    var assignment = new Array(points.length);
    var clusters = new Array(k);
    var movement = true;
    while (movement) {
      for (var i = 0; i < points.length; i++) {
         assignment[i] = closestCentroid(points[i], centroids);
      }
      movement = false;
    }
    return clusters;
};
```

Finally, by selecting a cluster at a time, you will recalculate the centroids and with these repeat the whole process until you get always the same values. First, as you can see in Listing 24-23, you make an iteration through the iterator `j` to analyze a cluster at a time. Inside of it, based on the contents of the `assignment` array, you fill the `assigned` array with all data points belonging to the cluster. These values serve you for the calculation of the new centroid defined in the `newCentroid` variable. To determine its new coordinates [x, y], you sum all x and y values, respectively, of all points of the cluster. These amounts are then divided by the number of points, so the x and y values of the new centroid are nothing more than the averages of all the coordinates.

To do all this, you need to implement a double iteration (two `for()` loops) with the `g` and `i` iterators. The iteration on `g` allows you to work on a coordinate at a time (first x, then y, and so on), while the iteration on `i` allows you to sum point after point in order to make the summation.

If the new centroids differ from the previous ones, then the assignment of the various data points to clusters repeats again, and the cycle begins again (steps 3 and 4 of the algorithm).

Listing 24-23. kmeans.js

```
function kmeans(points, k) {
    ...
    while (movement) {
      for (var i = 0; i < points.length; i++) {
         assignment[i] = closestCentroid(points[i], centroids);
      }
      movement = false;
      for (var j = 0; j < k; j++) {
         var assigned = [];
         for (var i = 0; i < assignment.length; i++) {
            if (assignment[i] == j) {
               assigned.push(points[i]);
            }
         }
         if (!assigned.length) {
            continue;
         }
         var centroid = centroids[j];
         var newCentroid = new Array(centroid.length);
         for (var g = 0; g < centroid.length; g++) {
            var sum = 0;
```

```
            for (var i = 0; i < assigned.length; i++) {
               sum += assigned[i][g];
            }
            newCentroid[g] = sum / assigned.length;
            if (newCentroid[g] != centroid[g]) {
               movement = true;
            }
         }
         centroids[j] = newCentroid;
         clusters[j] = assigned;
      }
    }
    return clusters;
};
```

Applying the Cluster Analysis to the Scatterplot

Having concluded the JavaScript code for the clustering analysis, it is time to come back to the web page. As you did for the example of the trendlines, you will use the code of the scatterplot as shown in Listing 24-24. This is the starting point on which you make the various changes and additions needed to integrate the cluster analysis.

Listing 24-24. ch24_03.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<script src="http://d3js.org/d3.v3.js"></script>
<style>
body {
    font: 16px sans-serif;
}
.axis path, .axis line {
    fill: none;
    stroke: #000;
    shape-rendering: crispEdges;
}
</style>
</head>
<body>
<script type="text/javascript">
var margin = {top: 70, right: 20, bottom: 40, left: 40},
    w = 500 - margin.left - margin.right,
    h = 400 - margin.top - margin.bottom;
var color = d3.scale.category10();
var x = d3.scale.linear()
    .range([0, w]);
var y = d3.scale.linear()
    .range([h, 0]);
var xAxis = d3.svg.axis()
    .scale(x)
```

```
    .orient("bottom");
var yAxis = d3.svg.axis()
    .scale(y)
    .orient("left");
var svg = d3.select("body").append("svg")
    .attr("width", w + margin.left + margin.right)
    .attr("height", h + margin.top + margin.bottom)
    .append("g")
    .attr("transform", "translate(" +margin.left+ "," +margin.top+ ")");
d3.tsv("data_09.tsv", function(error, data) {
    data.forEach(function(d) {
       d.time = +d.time;
       d.intensity = +d.intensity;
    });
    x.domain(d3.extent(data, function(d) { return d.time; })).nice();
    y.domain(d3.extent(data, function(d) { return d.intensity; })).nice();
    svg.append("g")
       .attr("class", "x axis")
       .attr("transform", "translate(0," + h + ")")
       .call(xAxis);
    svg.append("text")
       .attr("class", "label")
       .attr("x", w)
       .attr("y", h + margin.bottom - 5)
       .style("text-anchor", "end")
       .text("Time [s]");
    svg.append("g")
       .attr("class", "y axis")
       .call(yAxis);
    svg.append("text")
       .attr("class", "label")
       .attr("transform", "rotate(-90)")
       .attr("y", 6)
       .attr("dy", ".71em")
       .style("text-anchor", "end")
       .text("Intensity");
    var legend = svg.selectAll(".legend")
       .data(color.domain())
       .enter().append("g")
       .attr("class", "legend")
       .attr("transform", function(d, i) {
          return "translate(0," + (i * 20) + ")";
       });
    legend.append("rect")
       .attr("x", w - 18)
       .attr("width", 18)
       .attr("height", 18)
       .style("fill", color);
    legend.append("text")
       .attr("x", w - 24)
       .attr("y", 9)
```

```
        .attr("dy", ".35em")
        .style("text-anchor", "end")
        .text(function(d) { return d; });
});
var title = d3.select("svg").append("g")
    .attr("transform", "translate(" + margin.left + "," + margin.top + ")")
    .attr("class","title");
title.append("text")
    .attr("x", (w / 2))
    .attr("y", -30)
    .attr("text-anchor", "middle")
    .style("font-size", "22px")
    .text("My Scatterplot");
</script>
</body>
</html>
```

First, you need to include the file `kmeans.js` you have just created in order to use the functions defined within (see Listing 24-25).

Listing 24-25. ch24_03.html

```
...
<meta charset="utf-8">
<script src="http://d3js.org/d3.v3.js"></script>
<script src="./kmeans.js"></script>
<style>
body {
    ...
```

Prepare an array which will hold the data to be analyzed and call it `myPoints`. Once this is done, you can finally add the call to the `kmean()` function, as shown in Listing 24-26.

Listing 24-26. ch24_03.html

```
d3.tsv("data_09.tsv", function(error, data) {
    var myPoints = [];
    data.forEach(function(d) {
       d.time = +d.time;
       d.intensity = +d.intensity;
       myPoints.push([d.time, d.intensity]);
    });
    var clusters = kmeans(myPoints, 3);
    x.domain(d3.extent(data, function(d) { return d.time; })).nice();
    y.domain(d3.extent(data, function(d) { return d.intensity; })).nice();
    ...
};
```

Finally, you modify the definition of the circle SVG elements so that these are represented in the basis of the results returned by the `kmeans()` function, as shown in Listing 24-27.

Listing 24-27. ch24_03.html

```
d3.tsv("data_09.tsv", function(error, data) {
    ...
    svg.append("text")
      .attr("class", "label")
      .attr("transform", "rotate(-90)")
      .attr("y", 6)
      .attr("dy", ".71em")
      .style("text-anchor", "end")
      .text("Intensity");

    for(var i = 0; i < 3; i = i + 1){
       svg.selectAll(".dot" + i)
          .data(clusters[i])
          .enter().append("circle")
          .attr("class", "dot")
          .attr("r", 5)
          .attr("cx", function(d) { return x(d[0]); })
          .attr("cy", function(d) { return y(d[1]); })
          .style("fill", function(d) { return color(i); });
    }

    var legend = svg.selectAll(".legend")
       .data(color.domain())
       .enter().append("g")
    ...
};
```

In Figure 24-8, you can see the representation of one of the possible results which could be obtained after a clustering analysis.

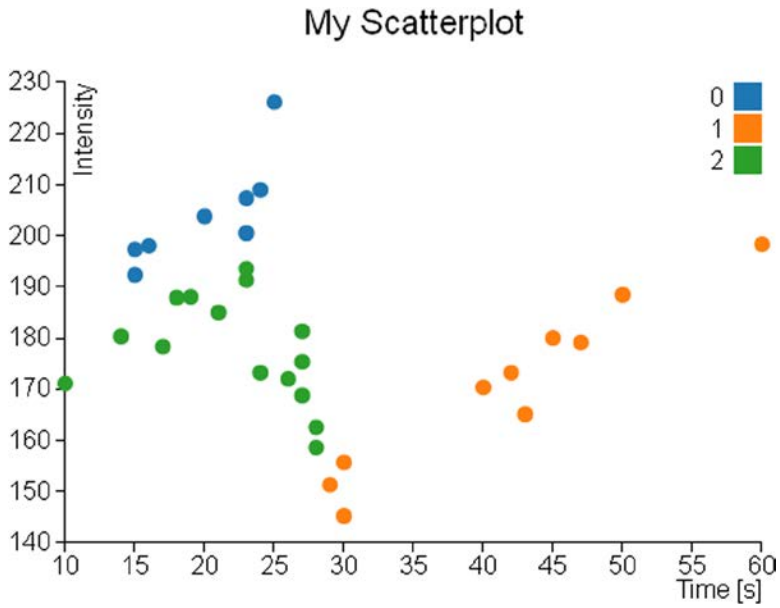


Figure 24-8. *The scatterplot shows one possible solution of the clustering analysis applied to the data in the TSV file*

Highlighting Data Points

Another functionality that you have not yet covered with the library D3, but you have seen with the jqPlot library (see Chapter 10) is **highlighting** and the events related to it. The D3 library even allows you to add this functionality to your charts and handle events in a way that is very similar to that seen with the jqPlot library.

The D3 library provides a particular function to activate or remove event listeners: the `on()` function. This function is applied directly to a selection by chaining method and generally requires two arguments: the *type* and the *listener*.

```
selection.on(type, listener);
```

The first argument is the type of event that you want to activate, and it is expressed as a string containing the event name (such as `mouseover`, `submit`, etc.). The second argument is typically made up of a function which acts as a listener and makes an operation when the event is triggered.

Based on all this, if you want to add the highlighting functionality, you need to manage two particular events: one is when the user hovers the mouse over a data point by highlighting it, and the other is when the user moves out the mouse from above the data point, restoring it to its normal state. These two events are defined in the D3 library as `mouseover` and `mouseout`. Now you have to join these events to two different actions. With `mouseover`, you will enlarge the volume of data points and you will increase the vividness of its color to further constrast it with the others. Instead, you will do the complete opposite with `mouseout`, restoring the color and size of the original data points.

Listing 24-28 shows the highlight functionality applied to the scatterplot code.

Listing 24-28. ch24_04.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
```

```
<script src="http://d3js.org/d3.v3.js"></script>
<style>
body {
    font: 16px sans-serif;
}
.axis path, .axis line {
    fill: none;
    stroke: #000;
    shape-rendering: crispEdges;
}
</style>
</head>
<body>
<script type="text/javascript">
var margin = {top: 70, right: 20, bottom: 40, left: 40},
    w = 500 - margin.left - margin.right,
    h = 400 - margin.top - margin.bottom;
var color = d3.scale.category10();
var x = d3.scale.linear()
    .range([0, w]);
var y = d3.scale.linear()
    .range([h, 0]);
var xAxis = d3.svg.axis()
    .scale(x)
    .orient("bottom");
var yAxis = d3.svg.axis()
    .scale(y)
    .orient("left");
var svg = d3.select("body").append("svg")
    .attr("width", w + margin.left + margin.right)
    .attr("height", h + margin.top + margin.bottom)
    .append("g")
    .attr("transform", "translate(" +margin.left+ "," +margin.top+ ")");
d3.tsv("data_09.tsv", function(error, data) {
    data.forEach(function(d) {
       d.time = +d.time;
       d.intensity = +d.intensity;
    });
    x.domain(d3.extent(data, function(d) { return d.time; })).nice();
    y.domain(d3.extent(data, function(d) { return d.intensity; })).nice();
    svg.append("g")
       .attr("class", "x axis")
       .attr("transform", "translate(0," + h + ")")
       .call(xAxis);
    svg.append("text")
       .attr("class", "label")
       .attr("x", w)
       .attr("y", h + margin.bottom - 5)
       .style("text-anchor", "end")
       .text("Time [s]");
```

```
    svg.append("g")
       .attr("class", "y axis")
       .call(yAxis);
    svg.append("text")
       .attr("class", "label")
       .attr("transform", "rotate(-90)")
       .attr("y", 6)
       .attr("dy", ".71em")
       .style("text-anchor", "end")
       .text("Intensity");

    var dots = svg.selectAll(".dot")
      .data(data)
      .enter().append("circle")
      .attr("class", "dot")
      .attr("r", 5)
      .attr("cx", function(d) { return x(d.time); })
      .attr("cy", function(d) { return y(d.intensity); })
      .style("fill", function(d) { return color(d.group); })
      .on("mouseover", function() { d3.select(this)
         .style("opacity",1.0)
         .attr("r", 15);
       })
       .on("mouseout", function() { d3.select(this)
          .style("opacity",0.6)
          .attr("r", 5);
       }) ;

    var legend = svg.selectAll(".legend")
       .data(color.domain())
       .enter().append("g")
       .attr("class", "legend")
       .attr("transform", function(d, i) {
          return "translate(0," + (i * 20) + ")";
       });
    legend.append("rect")
       .attr("x", w - 18)
       .attr("width", 18)
       .attr("height", 18)
       .style("fill", color);
    legend.append("text")
       .attr("x", w - 24)
       .attr("y", 9)
       .attr("dy", ".35em")
       .style("text-anchor", "end")
       .text(function(d) { return d; });
});
var title = d3.select("svg").append("g")
    .attr("transform", "translate(" + margin.left + "," + margin.top + ")")
    .attr("class","title");
title.append("text")
    .attr("x", (w / 2))
```

```
    .attr("y", -30)
    .attr("text-anchor", "middle")
    .style("font-size", "22px")
    .text("My Scatterplot");
</script>
</body>
</html>
```

Before loading the web page to see the result, you need to dull all the colors of the data points by setting the `opacity` attribute in the CSS styles, as shown in Listing 24-29.

Listing 24-29. ch24_04.html

```
<style>
body {
    font: 16px sans-serif;
}
.axis path,
.axis line {
    fill: none;
    stroke: #000;
    shape-rendering: crispEdges;
}
.dot {
    stroke: #000;
    opacity: 0.6;
}
</style>
```

Figure 24-9 shows one of many data points in the bubble chart in two different states. On the left you can see the data point in its normal state, while on the right it is highlighted.

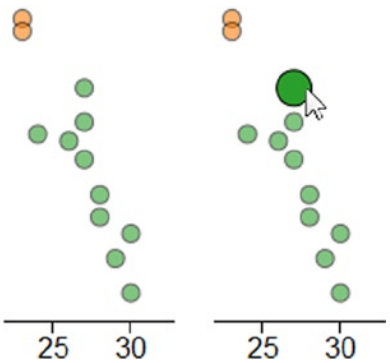


Figure 24-9. *A bubble assumes two states: normal on the left and highlighted when moused over on the right*

Bubble Chart

It is very easy to build a bubble chart by effecting only a few changes to the previous scatterplot example. First of all, you need to add a new column to your data. In this case (see Listing 24-30), you add bandwidth values as the last column to the `data_09.tsv` and you save it as `data_10.tsv`.

Listing 24-30. data_10.tsv

```
time    intensity    group    bandwidth
10    171.11    Exp1    20
14    180.31    Exp1    30
17    178.32    Exp1    10
42    173.22    Exp3    40
30    145.22    Exp2    35
30    155.68    Exp3    80
23    200.56    Exp2    10
15    192.33    Exp1    30
24    173.22    Exp2    10
20    203.78    Exp2    20
18    187.88    Exp1    60
45    180.00    Exp3    10
27    181.33    Exp2    40
16    198.03    Exp1    30
47    179.11    Exp3    20
27    175.33    Exp2    30
28    162.55    Exp2    10
24    208.97    Exp1    10
23    200.47    Exp1    10
43    165.08    Exp3    10
27    168.77    Exp2    20
23    193.55    Exp2    50
19    188.04    Exp1    10
40    170.36    Exp3    40
21    184.98    Exp2    20
15    197.33    Exp1    30
50    188.45    Exp3    10
23    207.33    Exp1    10
28    158.60    Exp2    10
29    151.31    Exp2    30
26    172.01    Exp2    20
23    191.33    Exp1    10
25    226.11    Exp1    10
60    198.33    Exp3    10
```

Now you have a third parameter in the list of data corresponding to the new column bandwidth. This value is expressed by a number, and in order to read it as such you need to add the `bandwidth` variable to the parsing of data, as shown in Listing 24-31. You must not forget to replace the name of the TSV file with `data_10.tsv` in the `tsv()` function.

Listing 24-31. ch24_05.html

```
d3.tsv("data_10.tsv", function(error, data) {
    var myPoints = [];
    data.forEach(function(d) {
       d.time = +d.time;
       d.intensity = +d.intensity;
       d.bandwidth = +d.bandwidth;
       myPoints.push([d.time, d.intensity]);
    });
    ...
});
```

Now you can turn all the dots into circular areas just by increasing their radius, since they are already set as SVG element `<circle>` as shown in Listing 24-32 The radii of these circles must be proportional to the bandwidth value, which therefore can be directly assigned to the `r` attribute. The 0.4 value is a correction factor which fits the bandwidth values to be very well represented in the bubble chart (in other cases, you will need to use other values as a factor).

Listing 24-32. ch24_05.html

```
d3.tsv("data_10.tsv", function(error, data) {
    ...
  svg.append("text")
      .attr("class", "label")
      .attr("transform", "rotate(-90)")
      .attr("y", 6)
      .attr("dy", ".71em")
      .style("text-anchor", "end")
      .text("Intensity");

    svg.selectAll(".dot")
       .data(data)
       .enter().append("circle")
       .attr("class", "dot")
       .attr("r", function(d) { return d.bandwidth * 0.4 })
       .attr("cx", function(d) { return x(d.time); })
       .attr("cy", function(d) { return y(d.intensity); })
       .style("fill", function(d) { return color(d.group); })
       .on("mouseover", function() { d3.select(this)
          .style("opacity",1.0)
          .attr("r", function(d) { return d.bandwidth * 0.5 });
       })
       .on("mouseout", function() { d3.select(this)
          .style("opacity",0.6)
          .attr("r", function(d) { return d.bandwidth * 0.4 });
       });

var legend = svg.selectAll(".legend")
    ...
});
```

Last but not least, you need to update the title of the new chart as shown in Listing 24-33.

Listing 24-33. ch24_05.html

```
title.append("text")
    .attr("x", (w / 2))
    .attr("y", -30 )
    .attr("text-anchor", "middle")
    .style("font-size", "22px")
    .text("My Bubble Chart");
```

And Figure 24-10 will be the result.

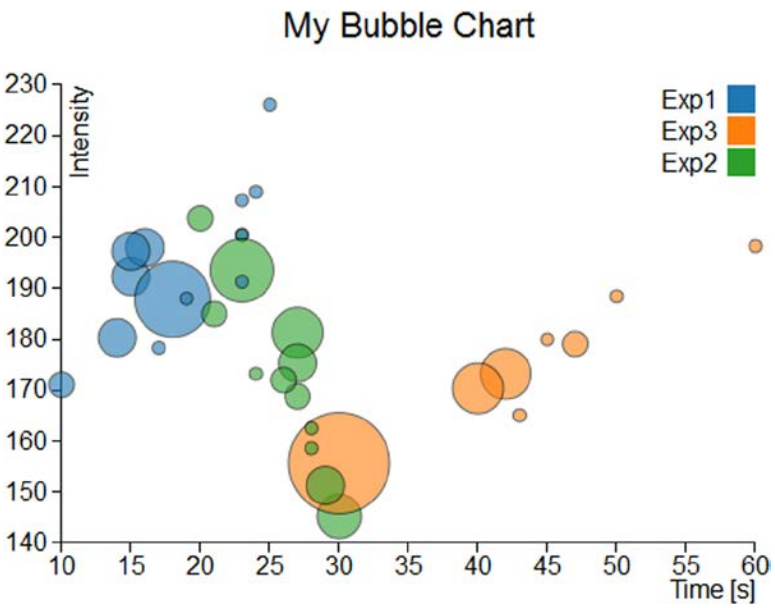


Figure 24-10. *A bubble chart*

Summary

In this chapter, you briefly saw how to generate **bubble charts** and **scatterplots** with the D3 library. Even here, you carried out the same type of charts which you saw in the second part of the book with the jqPlot library. Thus, you can get an idea of these two different libraries and of the respective approaches in the implementation of the same type of charts.

In the next chapter, you will implement a type of chart with which you still have not dealt with in the book: **radar charts**. This example of representation is not feasible with jqPlot, but it is possible to implement it thanks to D3 graphic elements. Thus, the next chapter will be a good example of how to use the potentialities of the D3 library to develop other types of charts which differ from those most commonly encountered.

CHAPTER 25

Radar Charts with D3

This chapter covers a type of chart that you have not yet read about: the radar chart. First you will get to know what it is, including its basic features, and how to create one using the SVG elements provided by the D3 library.

You'll start by reading a small handful of representative data from a CSV file. Then, making reference to the data, you'll see how it is possible to implement all the components of a radar chart, step by step. In the second part of the chapter, you'll use the same code to read data from more complex file, in which both the number of series and the amount of data to be processed are greater. This approach is a fairly common practice when you need to represent a new type of chart from scratch. You begin by working with a simple but complete example, and then, once you've implemented the basic example, you'll extend it with more complex and real data.

Radar Chart

Radar charts are also known as web or spider charts, for the typical web structure they assume (see Figure 25-1). They are two-dimensional charts that enable you to represent three or more quantitative variables. They consist of a sequence of spokes, all having the same angle, with each spoke representing one of the variables. A point is represented on every spoke and the point's distance from the center is proportional to the magnitude of the given variable. Then a line connects the point reported on each spoke, thus giving the plot a web-like appearance. Without this connecting line, the chart would look more like a scanning radar.

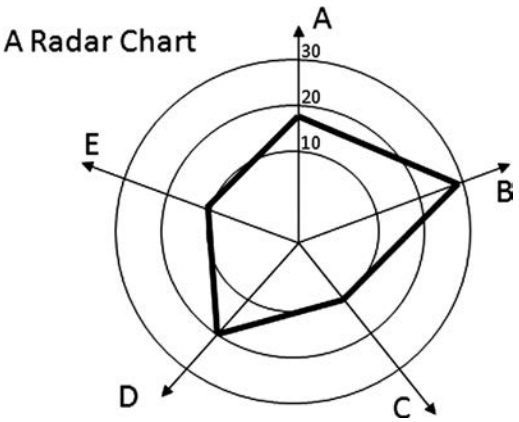


Figure 25-1. *A radar chart looks like a spider web*

Building Auto Scaling Axes

Copy the following data and save it in a file as `data_11.csv` (see Listing 25-1).

Listing 25-1. data_11.csv

```
section,set1,set2,
A,1,6,
B,2,7,
C,3,8,
D,4,9,
E,5,8,
F,4,7,
G,3,6,
H,2,5,
```

In Listing 25-2, you define the drawing area and the margins. You then create a color sequence with the `category10()` function.

Listing 25-2. ch25_01.html

```
var margin = {top: 70, right: 20, bottom: 40, left: 40},
    w = 500 - margin.left - margin.right,
    h = 400 - margin.top - margin.bottom;

var color = d3.scale.category20();
```

In the drawing area you just defined, you also have to define a specific area that can accommodate a circular shape, which in this case is the radar chart. Once you have defined this area, you define the radius like a Cartesian axis. In fact, each spoke on a radar chart is considered an axis upon which you place a variable. You therefore define a linear scale on the radius, as shown in Listing 25-3.

Listing 25-3. ch25_01.html

```
var  circleConstraint = d3.min([h, w]);
var  radius = d3.scale.linear()
    .range([0, (circleConstraint / 2)]);
```

You need to find the center of the drawing area. This is the center of the radar chart, and the point from which all the spokes radiate (see Listing 25-4).

Listing 25-4. ch25_01.html

```
var  centerXPos = w / 2 + margin.left;
var  centerYPos = h / 2 + margin.top;
```

Begin to draw the root element `<svg>`, as shown in Listing 25-5.

Listing 25-5. ch25_01.html

```
var svg = d3.select("body").append("svg")
    .attr("width", w + margin.left + margin.right)
    .attr("height", h + margin.top + margin.bottom)
    .append("g")
    .attr("transform", "translate(" + centerXPos + ", " + centerYPos + ")");
```

Now, as shown in Listing 25-6, you read the contents of the file with the `d3.csv()` function. You need to verify that the read values `set1` and `set2` are interpreted as numeric values. You also want to know which is the maximum value among all these, in order to define a scale that extends according to its value.

Listing 25-6. ch25_01.html

```
d3.csv("data_11.csv", function(error, data) {
    var maxValue = 0;
    data.forEach(function(d) {
       d.set1 = +d.set1;
       d.set2 = +d.set2;
       if(d.set1 > maxValue)
          maxValue = d.set1;
       if(d.set2 > maxValue)
          maxValue = d.set2;
    });
});
```

Once you know the maximum value of the input data, you set the full scale value equal to this maximum value multiplied by one and a half. In this case, instead of using the automatic generation of ticks on the axes, you have to define them manually. In fact, these ticks have a spherical shape and consequently, quite particular features. This example divides the range of the radius axis into five ticks. Once the values of the ticks are defined, you can assign a domain to the radius axis (see Listing 25-7).

Listing 25-7. ch25_01.html

```
d3.csv("data_11.csv", function(error, data) {
    ...
    data.forEach(function(d) {
       ...
    });

    var topValue = 1.5 * maxValue;
    var ticks = [];
    for(i = 0; i < 5;i += 1){
       ticks[i] = topValue * i / 5;
    }
    radius.domain([0,topValue]);
});
```

Now that you have all of the numerical values, we can embed some `<svg>` elements in order to design a radar grid that will vary in shape and value depending on the data entered (see Listing 25-8).

Listing 25-8. ch25_01.html

```
d3.csv("data_11.csv", function(error, data) {
    ...
    radius.domain([0,topValue]);

    var circleAxes = svg.selectAll(".circle-ticks")
       .data(ticks)
       .enter().append("g")
       .attr("class", "circle-ticks");
```

```
    circleAxes.append("svg:circle")
       .attr("r", function(d) {return radius(d);})
       .attr("class", "circle")
       .style("stroke", "#CCC")
       .style("fill", "none");

    circleAxes.append("svg:text")
       .attr("text-anchor", "middle")
       .attr("dy", function(d) {return radius(d)})
       .text(String);
});
```

You have created a structure of five <g> tags named `circle-ticks`, as you can see in Figure 25-2, each containing a <circle> element (which draws the grid) and a <text> element (which shows the corresponding numeric value).

```
<g transform="translate(260, 215)">
  <g class="circle-ticks">
      <circle class="circle" r="0" style="stroke: rgb(204, 204, 204); fill: none;">
      <text text-anchor="middle" dy="0">0</text>
    </g>
  <g class="circle-ticks">
  <g class="circle-ticks">
  <g class="circle-ticks">
  <g class="circle-ticks">
</g>
```

Figure 25-2. *FireBug shows how the* `circle-ticks` *are structured*

All of this code generates the circular grid shown in Figure 25-3.

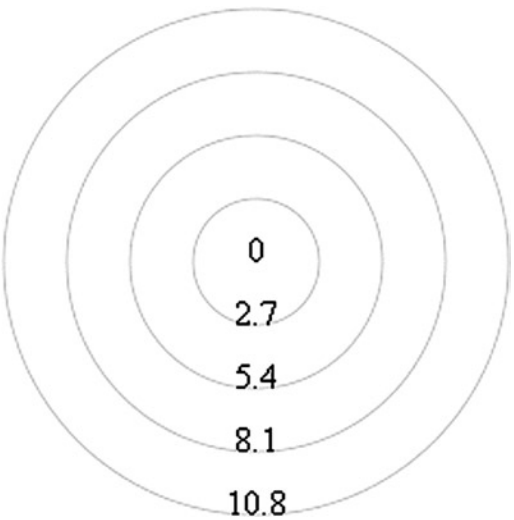


Figure 25-3. *The circular grid of a radar chart*

As you can see, the values reported on the tick will vary depending on the maximum value contained in the data.

Now is the time to draw the spokes, as many rays as there are lines in the `data_11.csv` file. Each of these lines corresponds to a variable, the name of which is entered in the first column of the file (see Listing 25-9).

Listing 25-9. ch25_01.html

```
d3.csv("data_11.csv", function(error, data) {
    ...
    circleAxes.append("svg:text")
      .attr("text-anchor", "middle")
      .attr("dy", function(d) {return radius(d)})
      .text(String);

    lineAxes = svg.selectAll('.line-ticks')
       .data(data)
       .enter().append('svg:g')
       .attr("transform", function (d, i) {
          return "rotate(" + ((i / data.length * 360) - 90) +
          ")translate(" + radius(topValue) + ")";
       })
       .attr("class", "line-ticks");

    lineAxes.append('svg:line')
       .attr("x2", -1 * radius(topValue))
       .style("stroke", "#CCC")
       .style("fill", "none");

    lineAxes.append('svg:text')
       .text(function(d) { return d.section; })
       .attr("text-anchor", "middle")
       .attr("transform", function (d, i) {
          return "rotate("+(90 - (i * 360 / data.length)) + ")";
      });
});
```

You now have spokes represented in the chart, as shown in Figure 25-4.

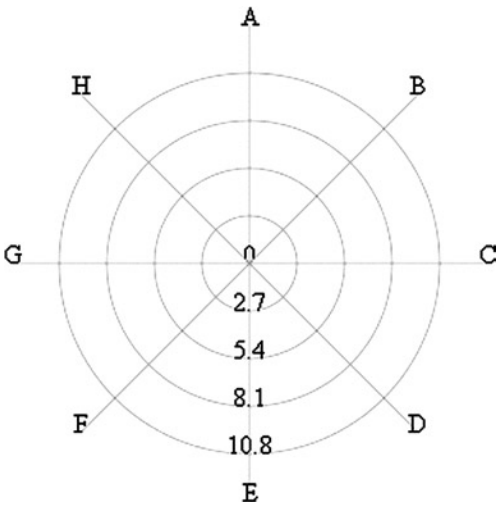


Figure 25-4. *The radial axes of a radar chart*

Adding Data to the Radar Chart

It is now time to consider the numeric columns in the file. Each column can be considered a series and each series must be assigned a color. You can define the series by taking the headers from the file and removing the first column. Then you can create the domain of colors according to the sequence of the series and then define the lines that draw them (see Listing 25-10).

Listing 25-10. ch25_01.html

```
d3.csv("data_11.csv", function(error, data) {
    ...
    lineAxes.append('svg:text')
      .text(function(d) { return d.section; })
      .attr("text-anchor", "middle")
      .attr("transform", function (d, i) {
          return "rotate("+(90-(i*360/data.length))+")";
    });

    var series = d3.keys(data[0])
       .filter(function(key) { return key !== "section"; })
       .filter(function(key) { return key !== ""; });

    color.domain(series);

    var lines = color.domain().map(function(name){
       return (data.concat(data[0])).map(function(d){
          return +d[name];
       });
    });
});
```

This is the content of the `series` arrays:

```
[ "set1", "set2" ]
```

And this is the content of the `lines` arrays:

```
[[1,2,3,...],[6,7,8,...]]
```

These lines will help you create the corresponding path elements and enable you to draw the trend of the series on the radar chart. Each series will pass through the values assumed in the various spokes (see Listing 25-11).

Listing 25-11. ch25_01.html

```
d3.csv("data_11.csv", function(error, data) {
    ...
    var lines = color.domain().map(function(name){
       return (data.concat(data[0])).map(function(d){
          return +d[name];
       });
});
    var sets = svg.selectAll(".series")
       .data(series)
       .enter().append("g")
       .attr("class", "series");

    sets.append('svg:path')
       .data(lines)
       .attr("class", "line")
       .attr("d", d3.svg.line.radial()
       .radius(function (d) {
          return radius(d);
       })
       .angle(function (d, i) {
          if (i == data.length) {
             i = 0;
          } //close the line
          return (i / data.length) * 2 * Math.PI;
       }))
       .data(series)
       .style("stroke-width", 3)
       .style("fill","none")
       .style("stroke", function(d,i){
          return color(i);
       });
});
```

You can also add a legend showing the names of the series (which are, actually, the headers of the columns) and a title placed on top in the drawing area, as shown in Listing 25-12.

Listing 25-12. ch25_01.html

```
d3.csv("data_11.csv", function(error, data) {
    ...
      .data(series)
      .style("stroke-width", 3)
      .style("fill","none")
      .style("stroke", function(d,i){
      return color(i);
      });

    var legend = svg.selectAll(".legend")
       .data(series)
       .enter().append("g")
       .attr("class", "legend")
       .attr("transform", function(d, i) {
         return "translate(0," + i * 20 + ")";
      });

    legend.append("rect")
       .attr("x", w/2 -18)
       .attr("y", h/2 - 60)
       .attr("width", 18)
       .attr("height", 18)
       .style("fill", function(d,i){ return color(i);});

    legend.append("text")
       .attr("x", w/2 -24)
       .attr("y", h/2 - 60)
       .attr("dy","1.2em")
       .style("text-anchor", "end")
       .text(function(d) { return d; });

});
var title = d3.select("svg").append("g")
    .attr("transform", "translate(" +margin.left+ "," +margin.top+ ")")
    .attr("class","title");

title.append("text")
    .attr("x", (w / 2))
    .attr("y", -30 )
    .attr("text-anchor", "middle")
    .style("font-size", "22px")
    .text("My Radar Chart");
```

Last but not least, you can add a CSS style class to rule the text style, as shown in Listing 25-13.

Listing 25-13. ch25_01.html

```
<style>
body {
    font: 16px sans-serif;
}
</style>
```

Figure 25-5 shows the resulting radar chart.

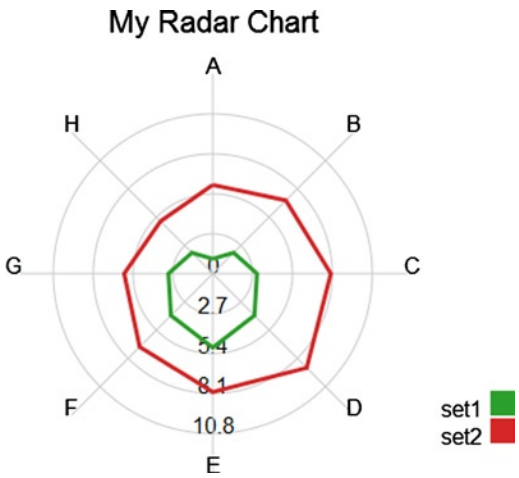


Figure 25-5. *A radar chart with two series*

Improving Your Radar Chart

If you followed along in the previous example, you should not have any problem adding more columns and rows to the radar chart. Open the last input data file, called `data_11.csv`, and add two more columns and rows. Save the file as `data_12.csv`, as shown in Listing 25-14.

Listing 25-14. data_12.csv

```
section,set1,set2,set3,set4,
A,1,6,2,10,
B,2,7,2,14,
C,3,8,1,10,
D,4,9,4,1,
E,5,8,7,2,
F,4,7,11,1,
G,3,6,14,2,
H,2,5,2,1,
I,3,4,5,2,
L,1,5,1,2,
```

You now have to replace the call to the `data11.csv` file with the `data12.csv` file in the `d3.csv()` function, as shown in Listing 25-15.

Listing 25-15. ch25_02.html

```
d3.csv("data_12.csv", function(error, data) {
    ...});
```

Figure 25-6 shows the result.

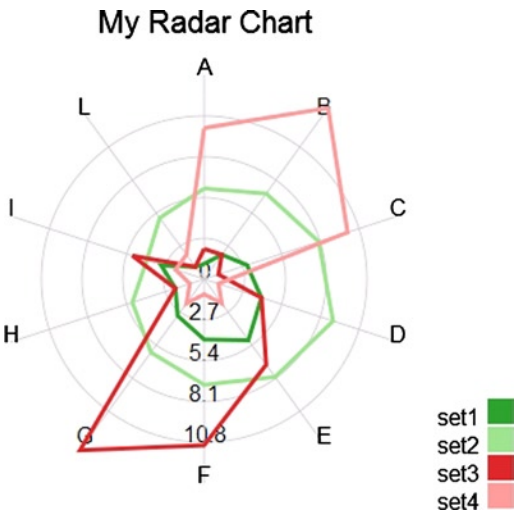


Figure 25-6. *A radar chart with four series*

Wow, it works! Ready to add yet another feature? So far, you've traced a line that runs through the various spokes to return circularly to the starting point; the trend now describes a specific area. You'll often be more interested in the areas of a radar chart that are delimited by the different lines than in the lines themselves. If you want to make this small conversion to your radar chart in order to show the areas, you need to add just one more path, as shown in Listing 25-16. This path is virtually identical to the one already present, only instead of drawing the line representing the series, this new path colors the area enclosed inside. In this example, you'll use the colors of the corresponding lines, but adding a bit of transparency, so as not to cover the underlying series.

Listing 25-16. ch25_02.html

```
d3.csv("data_12.csv", function(error, data) {
    ...
    var sets = svg.selectAll(".series")
       .data(series)
       .enter().append("g")
       .attr("class", "series");

    sets.append('svg:path')
       .data(lines)
       .attr("class", "line")
       .attr("d", d3.svg.line.radial()
```

```
        .radius(function (d) {
           return radius(d);
        })
        .angle(function (d, i) {
           if (i == data.length) {
              i = 0;
           }
           return (i / data.length) * 2 * Math.PI;
        }))
        .data(series)
        .style("stroke-width", 3)
        .style("opacity", 0.4)
        .style("fill",function(d,i){
           return color(i);
        })
        .style("stroke", function(d,i){
           return color(i);
        });

    sets.append('svg:path')
        .data(lines)
        .attr("class", "line")
        .attr("d", d3.svg.line.radial()
    ...
});
```

As you can see in Figure 25-7, you now have a radar chart with semi-transparent areas.

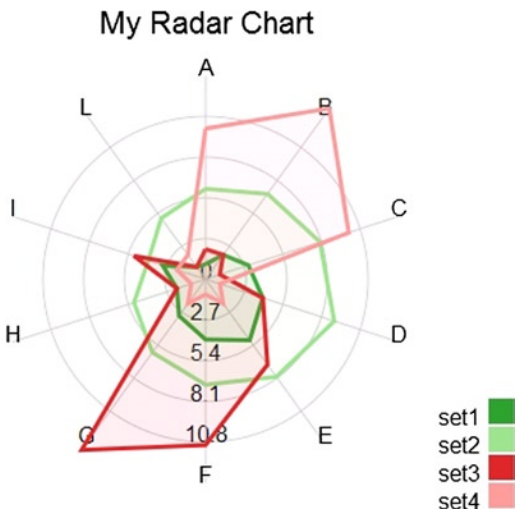


Figure 25-7. *A radar chart with color filled areas*

Summary

This chapter explained how to implement **radar charts**. This type of chart is not feasible with jqPlot and so this chapter was useful, in part to highlight the potential of the D3 library. It shows an example that helps you to understand how you can develop other charts that differ from the most commonly encountered types.

The next and final chapter concludes the book by considering two different cases. These cases are intended to propose, in a simplified way, the classic situations that developers have to face when they deal with real data. In the first example, you'll see how, using D3, it is possible to represent data that are generated or acquired in real time. You'll create a chart that's constantly being updated, always showing the current situation. In the second example, you'll use the D3 library to read the data contained in a database.

CHAPTER 26

Handling Live Data with D3

You have seen how to handle real-time charts with jqPlot, and in this chapter, you will implement the same example, using the D3 library. Indeed, you will create a line chart that displays the real-time values generated from a function that simulates an external source of data. The data will be generated continuously, and therefore the line chart will vary accordingly, always showing the latest situation.

In the second part of this chapter you will develop a chart that is slightly more complex. This time, you will be using an example in which the data source is a real database. First, you will implement a line chart that will read the data contained in an external file. Later, you will learn how to use the example to read the same data, but this time from the table of a database.

Real-Time Charts

You have a data source that simulates a function that returns random variations on the performance of a variable. These values are stored in an array that has the functions of a buffer, in which you want to contain only the ten most recent values. For each input value that is generated or acquired, the oldest one is replaced with the new one. The data contained in this array are displayed as a line chart that updates every 3 seconds. Everything is activated by clicking a button.

Let us start setting the bases to represent a line chart (to review developing line charts with the D3 library, see Chapter 20). First, you write the HTML structure on which you will build your chart, as shown in Listing 26-1.

Listing 26-1. ch26_01.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<script src="http://d3js.org/d3.v3.js"></script>
<style>
    //add the CSS styles here
</style>
<body>
<script type="text/javascript">
    // add the JavaScript code here
</script>
</body>
</html>
```

Now, you start to define the variables that will help you during the writing of the code, beginning with the management of the input data. As previously stated, the data that you are going to represent on the chart come from a function that generates random variations, either positive or negative, from a starting value. You decide to start from 10.

So, starting from this value, you receive from the random function a sequence of values to be collected within an array, which you will call `data` (see Listing 26-2). For now, you assign to it only inside the starting value (10). Given this array, to receive the values in real time, you will need to set a maximum limit, which, in this example, is eleven elements (0–10). Once filled, you will manage the array as a queue, in which the oldest item will be removed to make room for the new one.

Listing 26-2. ch26_01.html

```
<script type="text/javascript">
var data = [10];
w = 400;
h = 300;
margin_x = 32;
margin_y = 20;
ymax = 20;
ymin = 0;
y = d3.scale.linear().domain([ymin, ymax]).range([0 + margin_y, h - margin_y]);
x = d3.scale.linear().domain([0, 10]).range([0 + margin_x, w - margin_x]);
</script>
```

The values contained within the `data` array will be represented along the y axis, so it is necessary to establish the range of values that the tick labels will have to cover. Starting from the value of 10, you can decide, for example, that this range covers the values from 0 to 20 (later, you will ensure that the ticks correspond to a range of multiples of 5, i.e., 0, 5, 10, 15, and 20). Because the values to display are randomly generated, they will gradually assume values even higher than 20, and, if so, you will see the disappearance of the line over the top edge of the chart. What to do?

Because the main goal is to create a chart that redraws itself after acquiring new data, you will ensure that even the tick labels are adjusted according to the range covered by the values contained within the `data` array. To accomplish this, you need to define the y range with the `ymax` and `ymin` variables, with the `x` range covering the static range [0–10].

The `w` and `h` variables (width and height) define the size of the drawing area on which you will draw the line chart, whereas `margin_x` and `margin_y` allow you to adjust the margins.

Now, let us create the scalar vector graphics (SVG) elements that will represent the various parts of your chart. You start by creating the `<svg>` root and then define the x and y axes, as shown in Listing 26-3.

Listing 26-3. ch26_01.html

```
<script type="text/javascript">
...
y = d3.scale.linear().domain([ymin, ymax]).range([0 + margin_y, h - margin_y]);
x = d3.scale.linear().domain([0, 10]).range([0 + margin_x, w - margin_x]);

var svg = d3.select("body")
    .append("svg:svg")
    .attr("width", w)
    .attr("height", h);

var g = svg.append("svg:g")
    .attr("transform", "translate(0," + h + ")");
```

```
// draw the x axis
g.append("svg:line")
    .attr("x1", x(0))
    .attr("y1", -y(0))
    .attr("x2", x(w))
    .attr("y2", -y(0));

// draw the y axis
g.append("svg:line")
    .attr("x1", x(0))
    .attr("y1", -y(0))
    .attr("x2", x(0))
    .attr("y2", -y(25));
</script>
```

Next, you add the ticks and the corresponding labels on both axes and then the grid. Finally, you draw the line on the chart with the `line()` function (see Listing 26-4).

Listing 26-4. ch26_01.html

```
<script type="text/javascript">
...
g.append("svg:line")
    .attr("x1", x(0))
    .attr("y1", -y(0))
    .attr("x2", x(0))
    .attr("y2", -y(25));

//draw the xLabels
g.selectAll(".xLabel")
    .data(x.ticks(5))
    .enter().append("svg:text")
    .attr("class", "xLabel")
    .text(String)
    .attr("x", function(d) { return x(d) })
    .attr("y", 0)
    .attr("text-anchor", "middle");

// draw the yLabels
g.selectAll(".yLabel")
    .data(y.ticks(5))
    .enter().append("svg:text")
    .attr("class", "yLabel")
    .text(String)
    .attr("x", 25)
    .attr("y", function(d) { return -y(d) })
    .attr("text-anchor", "end");

//draw the x ticks
g.selectAll(".xTicks")
    .data(x.ticks(5))
    .enter().append("svg:line")
```

```
    .attr("class", "xTicks")
    .attr("x1", function(d) { return x(d); })
    .attr("y1", -y(0))
    .attr("x2", function(d) { return x(d); })
    .attr("y2", -y(0) - 5);

// draw the y ticks
g.selectAll(".yTicks")
    .data(y.ticks(5))
    .enter().append("svg:line")
    .attr("class", "yTicks")
    .attr("y1", function(d) { return -1 * y(d); })
    .attr("x1", x(0) + 5)
    .attr("y2", function(d) { return -1 * y(d); })
    .attr("x2", x(0))

//draw the x grid
g.selectAll(".xGrids")
    .data(x.ticks(5))
    .enter().append("svg:line")
    .attr("class", "xGrids")
    .attr("x1", function(d) { return x(d); })
    .attr("y1", -y(0))
    .attr("x2", function(d) { return x(d); })
    .attr("y2", -y(25));

// draw the y grid
g.selectAll(".yGrids")
    .data(y.ticks(5))
    .enter().append("svg:line")
    .attr("class", "yGrids")
    .attr("y1", function(d) { return -1 * y(d); })
    .attr("x1", x(w))
    .attr("y2", function(d) { return -y(d); })
    .attr("x2", x(0));

var line = d3.svg.line()
    .x(function(d,i) { return x(i); })
    .y(function(d) { return -y(d); })

</script>
```

To give a pleasing aspect to your chart, it is also necessary to define the Cascading Style Sheets (CSS) styles, as demonstrated in Listing 26-5.

Listing 26-5. ch26_01.html

```
<style>
path {
    stroke: steelblue;
    stroke-width: 3;
    fill: none;
}
```

```
line {
    stroke: black;
}
.xGrids {
    stroke: lightgray;
}
.yGrids {
    stroke: lightgray;
}
text {
    font-family: Verdana;
    font-size: 9pt;
}
</style>
```

Now, you add a button above the chart, ensuring that the `updateData()` function is activated when a user clicks it, as presented in Listing 26-6.

Listing 26-6. ch26_01.html

```
< body>
<div id="option">
    <input name="updateButton"
       type="button"
       value="Update"
       onclick="updateData()" />
</div>
```

Then, you implement the `getRandomInt()` function, which generates a random integer value between the minimum and maximum values (see Listing 26-7).

Listing 26-7. ch26_01.html

```
function getRandomInt (min, max) {
    return Math.floor(Math.random() * (max - min + 1)) + min;
};
```

Listing 26-8 shows the `updateData()` function, in which the value generated by the `getRandomInt()` function is added to the most recent value of the array to simulate variations on the trend. This new value is stored in the array, while the oldest value is removed; thus, the size of the array always remains the same.

Listing 26-8. ch26_01.html

```
function updateData() {
    var last = data[data.length-1];
    if(data.length > 10){
       data.shift();
    }
    var newlast = last + getRandomInt(-3,3);
    if(newlast < 0)
       newlast = 0;
    data.push(newlast);
};
```

If the new value returned by the `getRandomInt()` function is greater or less than the range represented on the y axis, you will see the line of data extending over the edges of the chart. To prevent this from occurring, you must change the interval on the y axis by varying the `ymin` and `ymax` variables and updating the y range with these new values, as shown in Listing 26-9.

Listing 26-9. ch26_01.html

```
function updateData() {
    ...
    if(newlast < 0)
       newlast = 0;
    data.push(newlast);

    if(newlast > ymax){
      ymin = ymin + (newlast - ymax);
      ymax = newlast;
      y = d3.scale.linear().domain([ymin, ymax])
         .range([0 + margin_y, h - margin_y]);
    }

    if(newlast < ymin){
       ymax = ymax - (ymin - newlast);
       ymin = newlast;
       y = d3.scale.linear().domain([ymin, ymax])
          .range([0 + margin_y, h - margin_y]);
    }
};
```

Because the new data acquired must then be redrawn, you will need to delete the invalid SVG elements and replace them with new ones. Let us do both with the tick labels and with the line of data (see Listing 26-10). Finally, it is necessary to repeat the refresh of the chart at a fixed time. Thus, using the `requestAnimFrame()` function, you can repeat the execution of the content of the `UpdateData()` function.

Listing 26-10. ch26_01.html

```
function updateData() {
    ...
    if(newlast < ymin){
       ymax = ymax - (ymin - newlast);
       ymin = newlast;
       y = d3.scale.linear().domain([ymin, ymax]).range([0 + margin_y, h - margin_y]);
    }
    var svg = d3.select("body").transition();
    g.selectAll(".yLabel").remove();
    g.selectAll(".yLabel")
       .data(y.ticks(5))
       .enter().append("svg:text")
       .attr("class", "yLabel")
       .text(String)
       .attr("x", 25)
       .attr("y", function(d) { return -y(d) })
       .attr("text-anchor", "end");
```

```
    g.selectAll(".line").remove();
    g.append("svg:path")
         .attr("class","line")
         .attr("d", line(data));

    window.requestAnimFrame = (function(){
       return  window.requestAnimationFrame ||
       window.webkitRequestAnimationFrame   ||
       window.mozRequestAnimationFrame      ||
          function( callback ){
              window.setTimeout(callback, 1000);
          };
    })();

    requestAnimFrame(setTimeout(updateData,3000));
    render();
};
```

Now, when the Update button has been clicked, from left a line begins to draw the values acquired by the function that generates random variations. Once the line reaches the right end of the chart, it will update every acquisition, showing only the last ten values (see Figure 26-1).

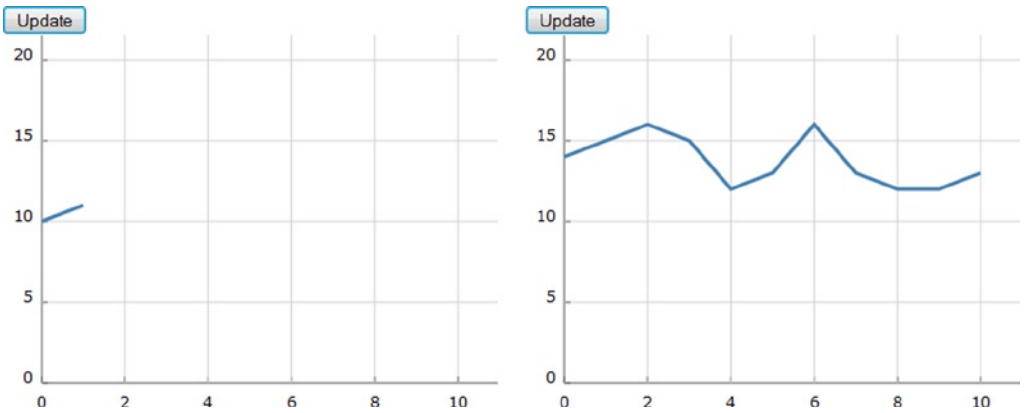


Figure 26-1. *A real-time chart with a start button*

Using PHP to Extract Data from a MySQL Table

Finally, the time has come to use data contained in a database, a scenario that is more likely to correspond with your daily needs. You choose MySQL as a database, and you use hypertext preprocessor (PHP) language to query the database and obtain the data in JavaScript Object Notation (JSON) format for it to be readable by D3. You will see that once a chart is built with D3, the transition to this stage is easy.

The following example is not intended to explain the use of PHP language or any other language, but to illustrate a typical and real case. The example shows how simple it is to interface all that you have learned with other programming languages. Often, languages such as Java and PHP provide an excellent interface for collecting and preparing data from their sources (a database, in this instance).

Starting with a TSV File

To understand more clearly the transition between what you already know and interfacing with PHP and databases, let us start with a case that should be familiar to you (see Listing 26-11). First, you write a tab-separated value (TSV) file with these series of data and save them as `data_13.tsv`.

Listing 26-11. data_13.tsv

```
day           income  expense
2012-02-12    52      40
2012-02-27    56      35
2012-03-02    31      45
2012-03-14    33      44
2012-03-30    44      54
2012-04-07    50      34
2012-04-18    65      36
2012-05-02    56      40
2012-05-19    41      56
2012-05-28    45      32
2012-06-03    54      44
2012-06-18    43      46
2012-06-29    39      52
```

■ **Note** Notice that the values in a TSV file are tab separated, so when you write or copy Listing 26-11, remember to check that there is only a tab character between each value.

Actually, as hinted at, you have already seen these data, although in slightly different form; these are the same data in the `data_03.tsv` file (see Listing 20-60 in Chapter 20). You changed the column `date` to `day` and modified the format for dates. Now, you must add the JavaScript code in Listing 26-12, which will allow you to represent these data as a multiseries line chart. (This code is very similar to that used in the section "Difference Line Chart" in Chapter 20; see that section for explanations and details about the content of Listing 26-12.)

Listing 26-12. ch26_02.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<script src="http://d3js.org/d3.v3.js"></script>
<style>
body {
....font: 10px verdana;
}
.axis path,
.axis line {
    fill: none;
    stroke: #333;
}
```

```
.grid .tick {
    stroke: lightgrey;
    opacity: 0.7;
}
.grid path {
    stroke-width: 0;
}
.line {
    fill: none;
    stroke: darkgreen;
    stroke-width: 2.5px;
}
.line2 {
    fill: none;
    stroke: darkred;
    stroke-width: 2.5px;
}
</style>
</head>
<body>
<script type="text/javascript">

var margin = {top: 70, right: 20, bottom: 30, left: 50},
    w = 400 - margin.left - margin.right,
    h = 400 - margin.top - margin.bottom;

var parseDate = d3.time.format("%Y-%m-%d").parse;

var x = d3.time.scale().range([0, w]);
var y = d3.scale.linear().range([h, 0]);

var xAxis = d3.svg.axis()
    .scale(x)
    .orient("bottom")
    .ticks(5);

var yAxis = d3.svg.axis()
    .scale(y)
    .orient("left")
    .ticks(5);

var xGrid = d3.svg.axis()
    .scale(x)
    .orient("bottom")
    .ticks(5)
    .tickSize(-h, 0, 0)
    .tickFormat("");

var yGrid = d3.svg.axis()
    .scale(y)
    .orient("left")
```

```
    .ticks(5)
    .tickSize(-w, 0, 0)
    .tickFormat("");

var svg = d3.select("body").append("svg")
    .attr("width", w + margin.left + margin.right)
    .attr("height", h + margin.top + margin.bottom)
    .append("g")
    .attr("transform", "translate(" + margin.left + "," + margin.top + ")");

var line = d3.svg.area()
    .interpolate("basis")
    .x(function(d) { return x(d.day); })
    .y(function(d) { return y(d["income"]); });

var line2 = d3.svg.area()
    .interpolate("basis")
    .x(function(d) { return x(d.day); })
    .y(function(d) { return y(d["expense"]); });

d3.tsv("data_13.tsv", function(error, data) {
    data.forEach(function(d) {
       d.day = parseDate(d.day);
       d.income = +d.income;
       d.expense = +d.expense;
    });

    x.domain(d3.extent(data, function(d) { return d.day; }));
    y.domain([
       d3.min(data, function(d) { return Math.min(d.income, d.expense); }),
       d3.max(data, function(d) { return Math.max(d.income, d.expense); })
    ]);

    svg.append("g")
       .attr("class", "x axis")
       .attr("transform", "translate(0," + h + ")")
       .call(xAxis);

    svg.append("g")
       .attr("class", "y axis")
       .call(yAxis);

    svg.append("g")
       .attr("class", "grid")
       .attr("transform", "translate(0," + h + ")")
       .call(xGrid);

    svg.append("g")
       .attr("class", "grid")
       .call(yGrid);
```

```
    svg.datum(data);

    svg.append("path")
       .attr("class", "line")
       .attr("d", line);

    svg.append("path")
       .attr("class", "line2")
       .attr("d", line2);
});

var labels = svg.append("g")
    .attr("class","labels");

labels.append("text")
    .attr("transform", "translate(0," + h + ")")
    .attr("x", (w-margin.right))
    .attr("dx", "-1.0em")
    .attr("dy", "2.0em")
    .text("[Months]");

labels.append("text")
    .attr("transform", "rotate(-90)")
    .attr("y", -40)
    .attr("dy", ".71em")
    .style("text-anchor", "end")
    .text("Millions ($)");

var title = svg.append("g")
    .attr("class", "title")

title.append("text")
    .attr("x", (w / 2))
    .attr("y", -30 )
    .attr("text-anchor", "middle")
    .style("font-size", "22px")
    .text("A Multiseries Line Chart");
</script>
</body>
</html>
```

With this code, you get the chart in Figure 26-2.

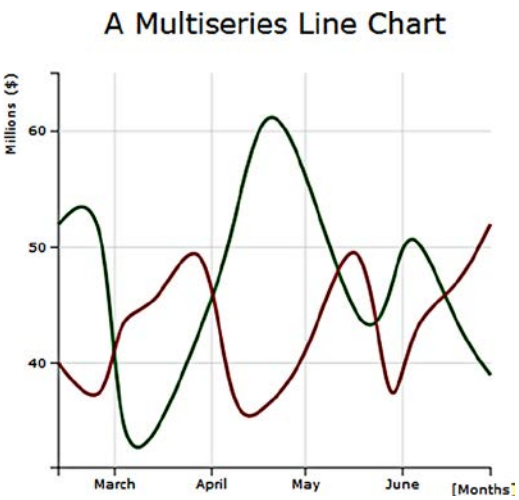


Figure 26-2. *A multiseries chart reading data from a TSV file*

Moving On to the Real Case

Now, let us move on to the real case, in which you will be dealing with tables in the database. For the data source, you choose a table called **sales** in a test database in MySQL.After you have created a table with this name, you can fill it with data executing an SQL sequence (see Listing 26-13).

Listing 26-13. sales.sql

```
insert into sales
values ('2012-02-12', 52, 40);
insert into sales
values ('2012-02-27', 56, 35);
insert into sales
values ('2012-03-02', 31, 45);
insert into sales
values ('2012-03-14', 33, 44);
insert into sales
values ('2012-03-30', 44, 54);
insert into sales
values ('2012-04-07', 50, 34);
insert into sales
values ('2012-04-18', 65, 36);
insert into sales
values ('2012-05-02', 56, 40);
insert into sales
values ('2012-05-19', 41, 56);
insert into sales
values ('2012-05-28', 45, 32);
insert into sales
values ('2012-06-03', 54, 44);
```

```
insert into sales
values ('2012-06-18', 43, 46);
insert into sales
values ('2012-06-29', 39, 52);
```

Preferably, you ought to write the PHP script in a separate file and save it as `myPHP.php`. The content of this file is shown in Listing 26-14.

Listing 26-14. myPHP.php

```
<?php
    $username = "dbuser";
    $password = "dbuser";
    $host = "localhost";
    $database = "test";

    $server = mysql_connect($host, $username, $password);
    $connection = mysql_select_db($database, $server);
    $myquery = "SELECT * FROM sales";
    $query = mysql_query($myquery);

    if ( ! $myquery ) {
       echo mysql_error();
       die;
    }

    $data = array();
    for ($x = 0; $x < mysql_num_rows($query); $x++) {
       $data[] = mysql_fetch_assoc($query);
    }
    echo json_encode($data);
    mysql_close($server);
?>
```

Generally, a PHP script is recognizable by its enclosure in special start and end processing instructions: `<?php` and `?>`. This short but powerful and versatile snippet of code is generally used whenever we need to connect to a database. Let us go through it and look at what it does.

In this example, `dbuser` has been chosen as user, with `dbuser` as password, but these values will depend on the database you want to connect to. The same applies to the database and hostname values. Thus, in order to connect to a dabase, you must first define a set of identifying variables, as shown in Listing 26-15.

Listing 26-15. myPHP.php

```
$username = "homedbuser";
$password = "homedbuser";
$host = "localhost";
$database="homedb";
```

Once you have defined them, PHP provides a set of already implemented functions, in which you have only to pass these variables as parameters to make a connection with a database. In this example, you need to call the `mysql_connect()` and `myqsl_select_db()` functions to create a connection with a database without defining anything else (see Listing 26-16).

Listing 26-16. myPHP.php

```
$server = mysql_connect($host, $username, $password);
$connection = mysql_select_db($database, $server);
```

Even for entering SQL queries, PHP proves to be a truly practical tool. Listing 26-17 is a very simple example of how to make an SQL query for retrieving data from the database. If you are not familiar with SQL language, a query is a declarative statement addressed to a particular database in order to obtain the desired data contained inside. You can easily recognize a query, as it consists of a `SELECT` statement followed by a FROM statement and almost always a `WHERE` statement at the end.

■ **Note** If you have no experience in SQL language and want to do a bit of practicing without having to install the database and everything else, I suggest visiting this web page from the w3schools web site: `www.w3schools.com/sql`. In it, you'll find complete documentation on the commands, with many examples and even the ability to query a database on evidence provided by the site by embedding an SQL test query (see the section "Try It Yourself").

In this simple example, the `SELECT` statement is followed by `'*'`, which means that you want to receive the data in all the columns contained in the table specified in the `FROM` statement (`sales`, in this instance).

Listing 26-17. myPHP.php

```
$myquery = "SELECT * FROM sales";
$query = mysql_query($myquery);
```

Once you have made a query, you need to check if it was successful and handle the error if one occurs (see Listing 26-18).

Listing 26-18. myPHP.php

```
if ( ! $query ) {
    echo mysql_error();
    die;
}
```

If the query is successful, then you need to handle the returned data from the query. You place these values in an array you call `$data`, as illustrated in Listing 26-19. This part is very similar to the `csv()` and `tsv()` functions in D3, only instead of reading line by line from a file, it is reading them from a table retrieved from a database. The `mysql_num_rows()` function gives the number of rows in the table, similar to the length() function of JavaScript used in `for()` loops. The `mysql_fetch_assoc()` function assigns the data retrieved from the query to the data array, line by line.

Listing 26-19. myPHP.php

```
$data = array();
for ($x = 0; $x < mysql_num_rows($query); $x++) {
    $data[] = mysql_fetch_assoc($query);
}
echo json_encode($data);
```

The key to the script is the call to the PHP `json_encode()` method, which converts the data format into JSON and then, with echo, returns the data, which D3 will parse. Finally, you must close the connection to the server, as demonstrated in Listing 26-20.

Listing 26-20. myPHP.php

```
mysql_close($server);
```

Now, you come back to the JavaScript code, changing only one row (yes, only one!) (see Listing 26-21). You replace the `tsv()` function with the `json()` function, passing directly the PHP file as argument.

Listing 26-21. ch26_02b.html

```
d3.json("myPHP.php", function(error, data) {
//d3.tsv("data_03.tsv", function(error, data) {
    data.forEach(function(d) {
       d.day = parseDate(d.day);
       d.income = +d.income;
    d.expense = +d.expense;
});
```

In the end, you get the same chart (see Figure 26-3).

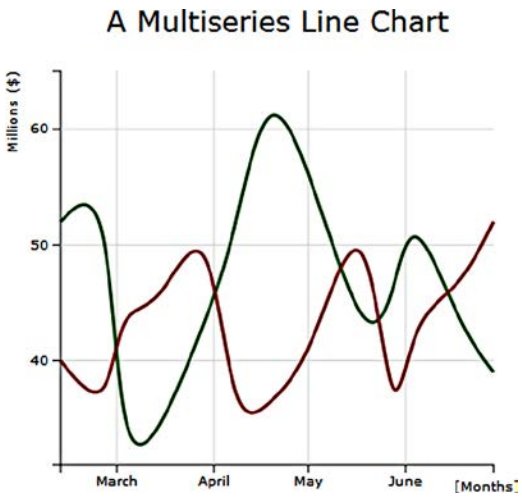


Figure 26-3. *A multiseries chart obtaining data directly from a database*

Summary

This final chapter concluded by considering two different cases. In the first example, you saw how to create a web page in which it is possible to represent data you are generating or acquiring in real time. In the second example, you learned how to use the D3 library to read the data contained in a database.

Conclusion

With this chapter, you have come to the end of this book. I must say that despite the large number of topics covered, there are many others I would have liked to add. I hope that this book has made you better appreciate the world of data visualization and charts in particular. I also hope that the book has provided you with a good, basic knowledge of data visualization and that it proves to be a valuable aid for all the occasions in which you find yourself dealing with charts.

APPENDIX A

Guidelines for the Examples in the Book

This appendix provides guidelines on how to use XAMPP and Aptana Studios together to create a development environment on your PC that will allow you to develop, run, and fix the examples given in the book.

Installing a Web Server

Nowadays, on the Internet, you can easily find free software packages containing everything you need to set up a test environment for all your examples and for everything related to the web world in general.

These packages minimize the number of programs that need to be installed.More important, they may be acquired with a single installation. The packages generally consist of an Apache HTTP server; a MySQL database; and interpreters for the programming languages PHP, Perl, and Python. The most complete package is XAMPP (available for download at the Apache Friends web site [`www.apachefriends.org/en/index.html`]). XAMPP is totally free, and its key feature is that it is a cross-platform package (Windows, Linux, Solaris, MacOS). Furthermore, XAMPP also includes a Tomcat application server (for the programming language Java) and a FileZilla FTP server (for file transfer). Other solutions are platform specific, as suggested by the initial letter of their name:

- **WAMP** (Windows)
- **MAMP** (MacOS)
- **LAMP** (Linux)
- **SAMP** (Solaris)
- **FAMP** (FreeBSD)

In fact, XAMPP is an acronym; its letters stand for the following terms:

- **X**, for the operating system
- **A**, for Apache, the web server
- **M**, for MySQL, the database management system
- **P**, for PHP, Perl, or Python, the programming languages

Thus, choose the web server solution that best fits your platform, and install it on your PC.

Installing Aptana Studio IDE

Once the Web server has been installed, it is necessary to install an integrated development environment (IDE), which you need to develop your JavaScript code. In this appendix, you will install Aptana Studio as your development environment.

Visit the Aptana site (`www.aptana.com`), and click the Products tab for the Aptana Studio 3 software (at the time of writing, the most recent version is 3.4.2). Download the stand-alone edition (with the Eclipse IDE already integrated): `Aptana_Studio_3_Setup_3.4.2.exe`.

After the download is complete, launch the executable file to install the Aptana Studio IDE. At the end of the installation, in launching the application, you should see the workbench opening, as shown in Figure A-1.

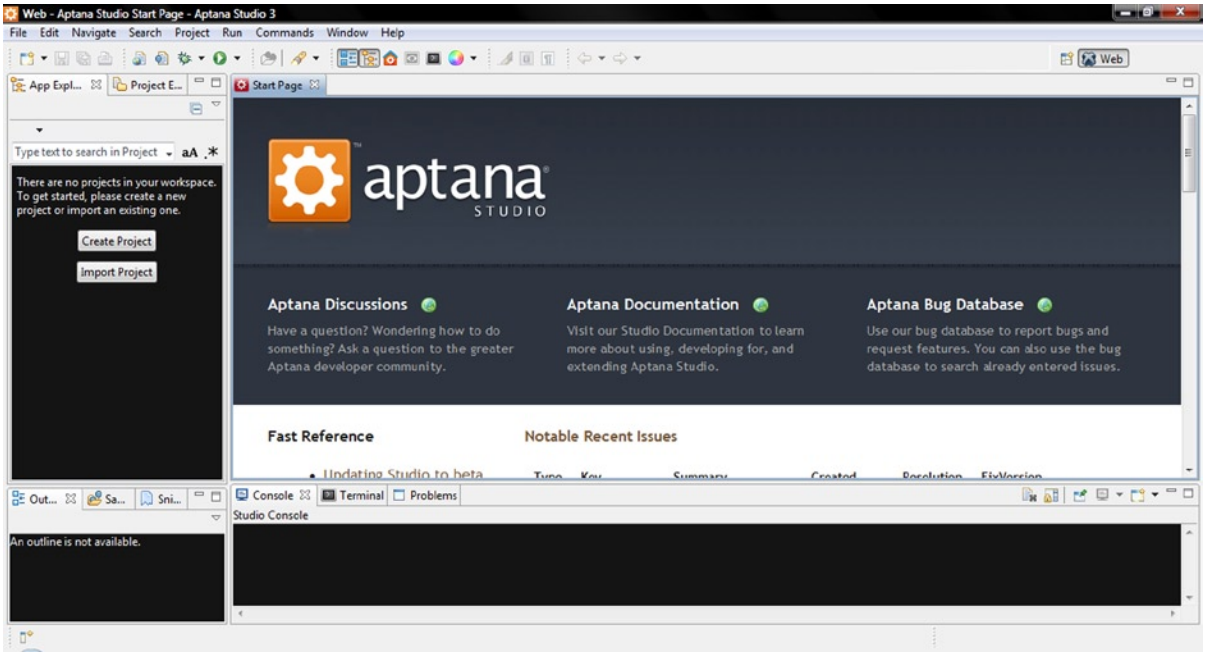

Web - Aptana Studio Start Page - Aptana Studio 3
File Edit Navigate Search Project Run Commands Window Help
Web
App Expl...
Project E...
Start Page
Type text to search in Project
There are no projects in your workspace.
To get started, please create a new project or import an existing one.
Create Project
Import Project
aptana
STUDIO
Aptana Discussions
Have a question? Wondering how to do something? Ask a question to the greater Aptana developer community.
Aptana Documentation
Visit our Studio Documentation to learn more about using, developing for, and extending Aptana Studio.
Aptana Bug Database
Use our bug database to report bugs and request features. You can also use the bug database to search already entered issues.
Fast Reference
Notable Recent Issues
Out...
Sa...
Sni...
An outline is not available.
Console
Terminal
Problems
Studio Console

Figure A-1. *The Aptana Studio IDE workbench*

During the installation of Aptana Studio, the software detects the various browsers and the web server installed and configures itself accordingly.

Setting the Aptana Studio Workspace

Before starting to develop the examples in the book, you must create a workspace. First, you should set the workspace on Aptana Studio, where the Web server document root is.

These are typical paths with XAMPP:

- Windows: `C:\xampp\htdocs`
- Linux: `/opt/lamp/htdocs`
- MacOS: `/Applications/XAMPP/xamppfiles/htdocs`

Whereas with WAMP, this is the path:

- `C:\WAMP\www`

Thus, select File ➤ Switch Workspace ➤ Other . . . from the menu. Then, insert the path of the web server document root in the field, as demonstrated in Figure A-2.

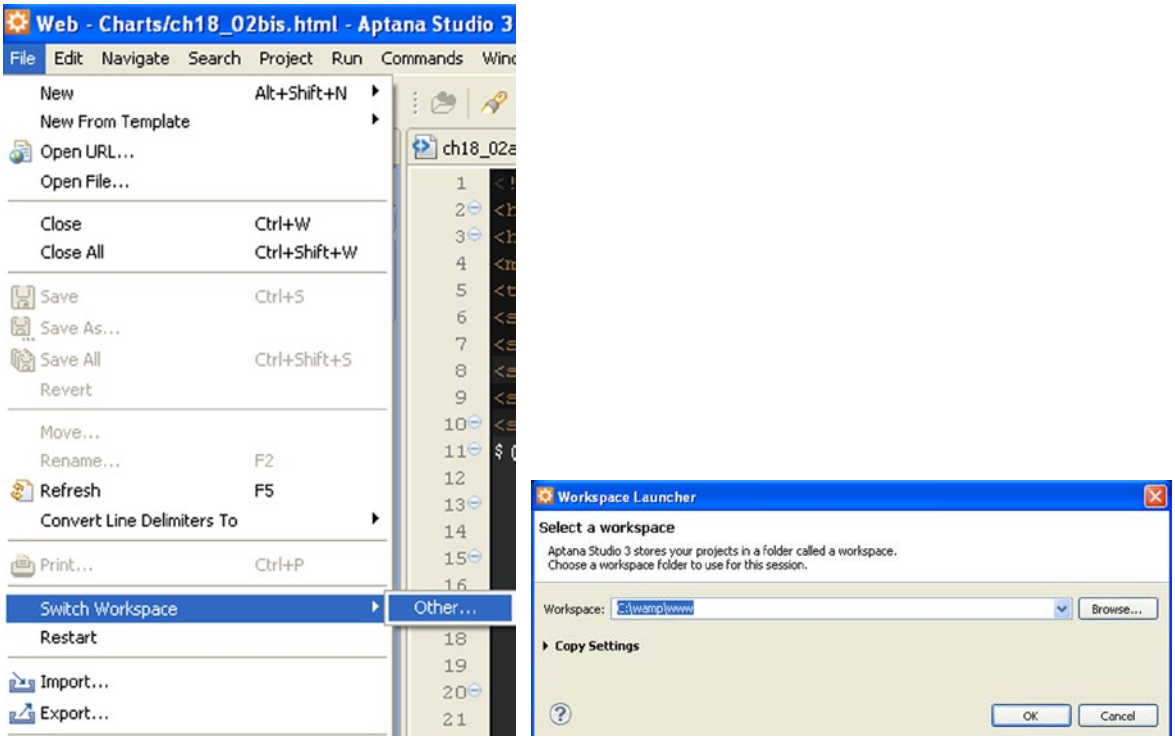


Figure A-2. *Setting the workspace on the document root*

Creating a Project

The next step in creating your workspace consists of creating a project in Aptana Studio:

1. Select New ➤ Web Project from the menu.
2. A window such as that shown in Figure A-3 appears. Select Default Project, and click Next.

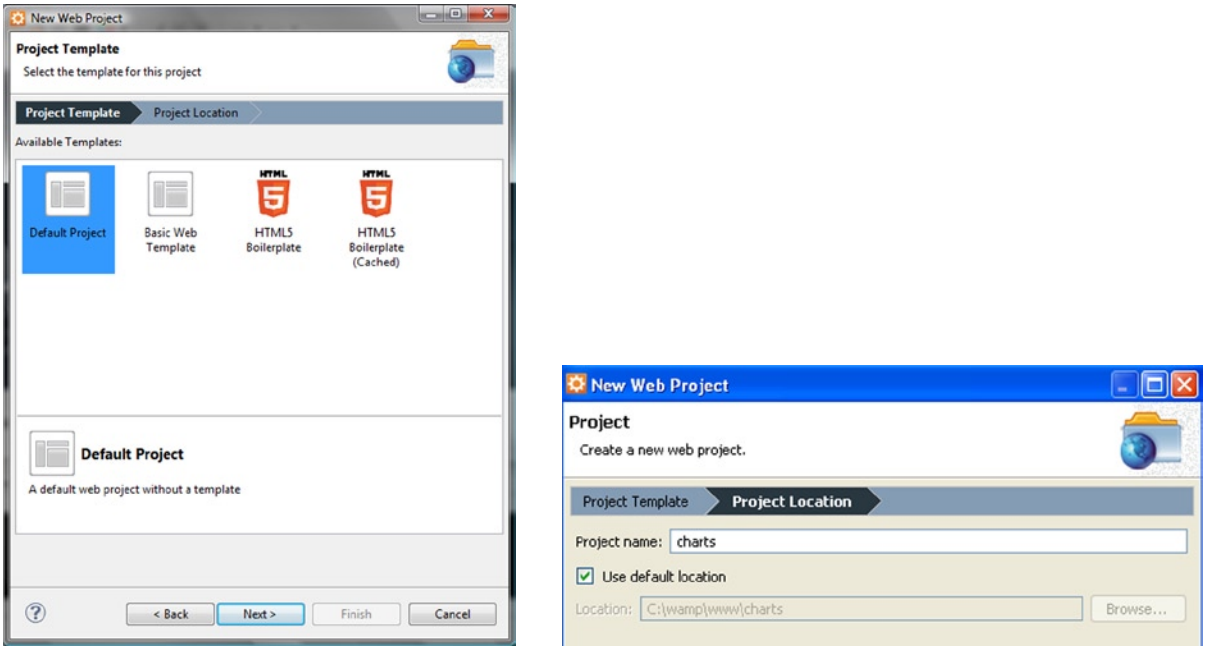


Figure A-3. *Creating a default project*

3. Insert "charts" as the name of the project. This will be the directory in the workspace in which you will write all the example files described in the book, using Aptana Studio.

Completing the Workspace

Once you have set the Aptana Studio workspace and created a project, you complete the workspace.

Let us open the document root directory and create a new directory, named `src`. Now, the workspace on which wyou will be working throughout the book is composed of two directories:

- `src`
- `charts`

The `src` directory should contain all the files related to libraries.

The `charts` directory should contain all HTML, images and Cascading Style Sheets (CSS) files related to the examples in the book (which is in fact a project). Each example file should be created in this directory (if you prefer to do things differently, that's fine, but it is important to take note of the different path reference in HTML pages in order to include the library files and images).

■ **Note** The source code that accompanies this book (available from the Source Code/Download area of the Apress web site [`www.apress.com`]) is practically already packaged in a workspace. With it, you will find two versions of the charts project: content delivery network (CDN) and local. The `charts_CDN` directory contains all the examples referring to libraries remotely distributed from CDN services. The `charts_local` directory offers all the examples referring to libraries found within the `src` directory.

Filling the src Directory with the Libraries

If you have chosen to develop HTML pages by referring to libraries locally, it is necessary to download all their files. These files will be collected in the `src` directory. This is a good approach, as you can develop several projects that will make use of the same libraries without having to copy them for each project.

The versions listed in this appendix are those used to implement the examples in the book. If you install other versions, there may be issues of incompatibility, or you may observe behavior different from that described.

jqPlot library version 1.0.8 (includes jQuery library version 1.9.1)

1. Visit the jqPlot web site (https://bitbucket.org/cleonello/jqplot/downloads/), and download the compressed file (.zip, .tar.gz or tar.bz2) for the library: `jquery.jqplot.1.0.8r1250`.
2. Extract all content. You should get a directory named `dist`, containing the following subdirectories and files:
   - `doc`
   - `examples`
   - `plugins`
   - A series of files (`jquery.min.js`, `jquery.jqplot.min.js`, and so on)
3. Copy the set of files and the `plugins` directory, and place in `src`.

jquery UI library version 1.10.3, with the smoothness theme

1. Visit the JQuery user interface library (jQuery UI) site (http://jqueryui.com/themeroller/), and download the library from ThemeRoller, with the smoothness theme: `jquery-ui-1.10.3.custom.zip`.
2. Extract all content. You should get a directory named `jquery-ui-1.10.3.custom`, with the following directories inside:
   - `css`
   - `js`
   - `development-bundle`
3. Copy the `css` and `js` directories, and place in `src`.

D3 library version 3

1. Visit the D3 site (http://d3js.org), and download the library: `d3.v3.zip`.
2. Extract all content directly, and place in the `src` directory. Now, you should have two new files in the `src` directory:
   - `d3.v3.js`
   - `d3.v3.min.js`

Highcharts library version 3.0.5

1. Visit the Highcharts site (www.highcharts.com), and download the library: `Highcharts-3.0.5.zip`.
2. Extract all content. You get a directory with several directories inside.
3. Copy only the `js` directory, and place in `src`.

You have thus obtained the `src` directory, which should contain the subdirectories and files shown in Figure A-4.

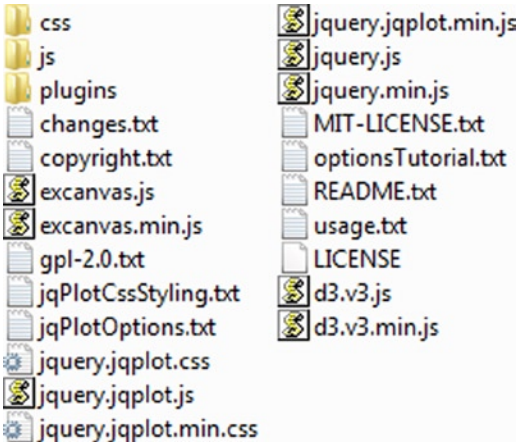


Figure A-4. *The files and subdirectories contained in the* `src` *directory*

■ **Note** By convention you are developing the examples in the `charts` directory. If you want to do otherwise, you need to consider the new path when you will include the other files in a web page.

If you are developing the HTML page inside the `charts` directory, you need to use the following code:

```
<script type="text/javascript" src="../src/jquery.min.js"></script>
```

In contrast, if you prefer to develop it directly, in the document root, you use this:

```
<script type="text/javascript" src="src/jquery.min.js"></script>
```

In short, it is important to take the path of the file you are including into account, with respect to the page you are implementing.

Running the Examples

Once you have created or copied an HTML file in the workspace, to run it in Aptana Studio IDE, select Run ➤ Run from the menu, or click the Run button on the toolbar (see Figure A-5).

Figure A-5. *The Run button from the toolbar*

Immediately, your default browser will open, with the selected HTML page loaded.

Look at Run Configurations (see Figure A-6), selecting Run Configurations . . . from the context menu of the Run icon. Let us set, for example, `http://localhost/` as your base URL; to do so, you select the Append project name option, as shown. Then, you click the Apply button to confirm your settings.

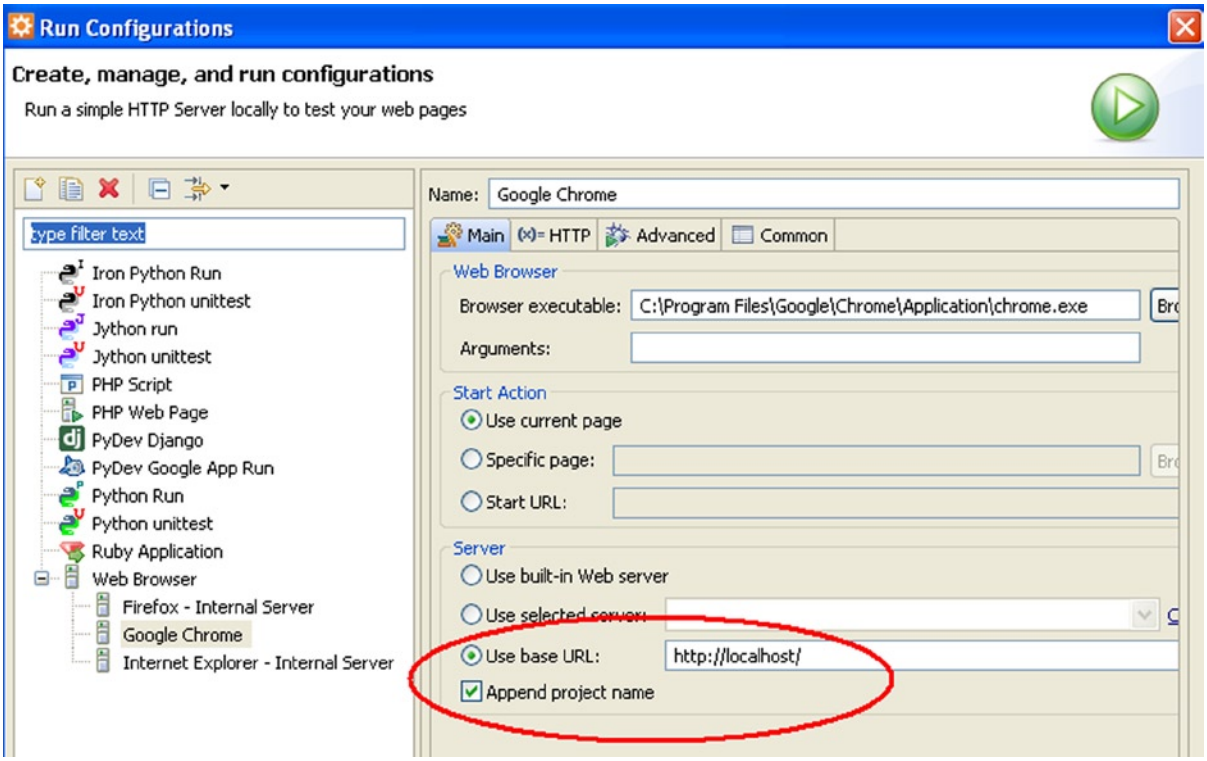


Figure A-6. *The run configuration for each browser must be set correctly*

Now, you have everything required to work easily on all the examples in the book.

Once you reach a certain familiarity with the Aptana IDE, you will find that it is an excellent environment for developing many other projects, both in JavaScript and in other programming languages (e.g., PHP).

And, now, have fun!

Summary

This appendix provides guidelines on how to use XAMPP and Aptana Studios together to create a development environment on your PC. The choice of using these applications is not mandatory, and many other solutions are possible; there are many applications available on the Internet for performing similar operations. But, if you wish to implement and quickly test the examples described in the book, this environment will prove a good choice.

APPENDIX B

jqPlot Plug-ins

This appendix shows the complete list of available plug-ins in the jqPlot distibution (see Table B-1). Not all these plug-ins have been treated in this book; for more information, please visit the jqPlot web site (`www.jqplot.com`).

Table B-1. *Available Plug-ins in the jqPlot Distribution (version 1.0.8)*

Name	Type	Description
$.jqplot.BarRenderer	Renderer	Draw a bar chart.
$.jqplot.BezierCurveRenderer	Renderer	Draw lines as stacked Bezier curves.
$.jqplot.BlockRenderer	Renderer	Draw an xy block chart. A block chart has data points displayed as colored squares, with a text label inside.
$.jqplot.BubbleRenderer	Renderer	Draw a bubble chart. A bubble chart has data points displayed as colored circles, with an optional text label inside.
$.jqplot.CanvasAxisLabelRenderer	Renderer	Draw axis labels, with a `canvas` element to support advanced features, such as rotated text. This renderer uses a separate rendering engine to draw the text on the canvas.
$.jqplot.CanvasAxisTickRenderer	Renderer	Draw axis ticks, with a `canvas` element to support advanced features, such as rotated text. This renderer uses a separate rendering engine to draw the text on the canvas.
$.jqplot.CanvasOverlay	Plug-in	Draw lines overlaying the chart.
$.jqplot.CanvasTextRenderer	Renderer	Modified version of the `canvastext.js` plug-in, written by Jim Studt (`http://jim.studt.net/canvastext/`).
$.jqplot.CategoryAxisRenderer	Renderer	Render a category style axis, with equal pixel spacing between y data values of a series.
$.jqplot.ciParser	Plug-in	A function to convert a custom JavaScript Object Notation (JSON) data object into jqPlot data format.
$.jqplot.Cursor	Plug-in	A class representing the cursor, as displayed on the plot.
$.jqplot.DateAxisRenderer	Renderer	Render an axis as a series of date values.
$.jqplot.DonutRenderer	Renderer	Draw a donut chart; x values, if present, are used as slice labels, and y values give slice size.
$.jqplot.Dragable	Plug-in	Make plotted points that the user can drag.

(*continued*)

Table B-1. (*continued*)

Name	Type	Description
$.jqplot.EnhancedLegendRenderer	Renderer	Draw **a legend** with advanced features.
$.jqplot.FunnelRenderer	Renderer	Draw a funnel chart; x values, if present, are used as labels, and y values give area size. Funnel charts draw a single series only.
$.jqplot.Highlighter	Plug-in	Highlight data points when they are moused over.
$.jqplot.Json2	Plug-in	Create a JSON object containing two methods: `stringify()` and `parse()`.
$.jqplot.LogAxisRenderer	Renderer	Render a logarithmic axis.
$.jqplot.MekkoAxisRenderer	Renderer	Used along with the *MekkoRenderer* plug-in; displays the y axis as a range from 0 to 1 (0 to 100 percent) and the x axis with a tick for each series, scaled to the sum of all the y values.
$.jqplot.MekkoRenderer	Renderer	Draw a Mekko-style chart that shows three-dimensional data on a two-dimensional graph.
$.jqplot.MeterGaugeRenderer	Renderer	Draw a meter gauge chart.
$.jqplot.Mobile	Plug-in	jQuery mobile virtual event support.
$.jqplot. OHLCRenderer	Renderer	Draw open-high-low-close, candlestick, and high-low-close charts.
$.jqplot.PieRenderer	Renderer	Draw a pie chart; x values, if present, are used as slice labels, and y values give slice size.
$.jqplot.PointLabels	Plug-in	Place labels at the data points.
$.jqplot.PyramidAxisRenderer	Renderer	Used along with the *PyramidRenderer* plug-in; displays the two x axes at the bottom and the y axis at the center.
$.jqplot.PyramidGridRenderer	Renderer	Used along with the *PyramidRenderer* plug-in; creates a grid on a `canvas` element.
$.jqplot.PyramidRenderer	Renderer	Draw a pyramid chart.
$.jqplot.Trendline	Plug-in	Automatically compute and draw trend lines for plotted data.

Index

A

accordion() function, 309
allowPointSelect property, 362
Animated multiseries line chart, 173–174
animateReplot property, 237–238
append() method, 381, 386
Area charts
  combined chart, 208
  multiple data arrays, 206
  multiseries line chart, 206–207
  seriesDefaults, 206
attr() operator, 379
Axes, line charts
  empty space, 156
  min and max properties, 157–158
  pad properties, 156–157
  ticks
    axesDefaults object, 161
    CanvasAxisTickRenderer, 161
    chart without a grid, 162
    directly defined ticks, 159–160
    horizontal grid lines, 162–163
    nonuniform, prefixed ticks, 160–161
    numberTicks property, 158–159
    percentage values on y axis, 163–164
  title addition and labels
    CanvasAxisLabelRenderer plug-in, 153–154
    listing code, 155
    properties, 154

B

Band charts, 208
Bar charts, 359. *See also* BarRenderer plug-in
  creation, 82
  data representation, 84
drawing
  category10() function, 450
  CDN service, 449
  CSS class attributes, 454
  d3.csv() function, 451
  d3.svg.axis(), 450
  data_04.csv, 449
  domain for axes, 452
  forEach() loop function, 452
  function(error,data) function, 454
  grid lines, 452–453
  rangeRoundBands function, 450
  simple bar chart, 455
  size and dimensions, 450
  SVG elements, 452
  title element, 453–454
  variables, 453
grouped
  code snippets, 468–469
  csv() function, 472
  data_05.csv, 468
  filter() function, 470
  legend addition, 474
  new domains, 471
  point labels, 473
  title addition, 472
  xo and x1 variables, 469–470
horizontal
  Firebug, 477
  margin addition, 475
  maximum value, 475
  positive and negative values, 475
  <rect> element insertion, 477
  scales with x axis and y axis, 476
  style class attributes, 478
  <svg> element creation, 476
HTML table, 81
jQuery css() function, 83

Bar charts, 359. *See also* BarRenderer plug-in (*cont.*)
row deletion, 82
stacked (*see* Stacked bar chart)
xDelta variable, 81
barMargin property, 224
BarRenderer plug-in
adding values, 225–226
animated plot, 237–238
barDirection property, 231
colors, 230
combination chart, 235–236
creation, 221
custom tool tips
bar chart, 255
jqplotDataUnhighlight event, 254
jqplot() function, 254–255
PNG file, 253
qplotDataHighlight event, 254
handling legends
CSS class, 249
custom legend, 251–253
default legend, 249
definition, 247
EnhancedLegendRenderer, 250
location attribute, 248
options, 247
stacked bar chart, 248
horizontal multiseries bar chart, 231
horizontal stacked chart, 234
jqlotDataClick event, 241–242, 245–247
jqPlotDataHighlight event, 244
jqplotDataMouseOver event, 243, 245
jqplotDataUnhighlight event, 244
jqplotRightClick event, 242–243
label property, 228
Marimekko chart, 238–240
multiseries bar chart, 229–230
negative values, 226–228
options, 229
space modification, 224
tick label rotation, 223–224
ticks array, 228
vertical stacked chart, 232–233
bind() function, 313
Block chart, 279–281
Bubble chart
bandwidth variable, 541
bubbleAlpha property, 278
BubbleRenderer, 276
circle SVG element, 542
data_10.tsv, 541
data display, 276
data identification, 276
jqplot() function, 278
label reporting, 276
options variable, 276–277
pay attention!, 276
rows of code, 277
selected state, foreground, 278
Sweden data, 277
title updation, 542
transparent effect, 279

C

Candlestick charts. *See also* Open-high-low-close (OHLC) chart
application, 267
box representation, 271
OHLC chart
dateAxisRenderer object, 268
DateAxisRenderer plug-in, 267
Dukascopy, 268
euro–US dollar exchange value, 268
OHLCRenderer, 267
with lines, 269
real bodies and shadows, 270
renderer property, 271
ticks array, 270–271
Canvas
colors, 63
definition, 61
document object model element, 62
JavaScript code, 62
rectangle, 64
translate() method, 63
two-dimensional drawing application programming interface, 62
web page, 64
CanvasAxisLabelRenderer plug-in, 154
Cascading style sheets (CSS), 121, 124, 249, 291
chart controls, 291
chart-title, 78
create dynamic text, HTML, 66
CSV file
axes and grid, 420
data drawing, 425
gradient generator, 47
grid background, 71
HTML, 46
input data addition, 404–405
jqPlot charts, 305
legend, 76
line chart (*see* Line charts, chart appearance, CSS)
table, 61
ticks and labels, 70
UI widgets, 38–39

Chart controls
check boxes, 287
"CheckAll/UncheckAll" feature, 299
custom legend, 301
EnhancedLegendRenderer plug-in, 301
jQuery methods, 299–300
list, 298
number of series, 298
command, 287
functionality, 287
radio button
accessing attributes, 292
CDN service, 289
change() function, 290
CSS style, 291
data selection, 291
first series representation, 289
HTML page, 288
jQuery $(document).ready() function, 289
multiseries line chart, 288
radio buttons, 287
sliders, 287
code section, 296
hexadecimal format, 296
jQuery Interface library, 294
numerical values, 295
RBG values, 297
RGB values, 294
Charting technology
canvas and SVG, 7
DevTools, 16–17
DOM, 9
elements
axis label, 2
legend, 4
point label and tooltip, 3
two-dimensional chart, 2
Firebug, 16
HTML5, 6
JavaScript
Aptana Studio, 10
arrays, 13
IDEs, 10
libraries, 11
objects, 15
online IDE jsFiddle, 11
running and debugging, 12
variable, 13
JSON, 18
types, 4
Chart-title, 78
closePath() function, 74
Combined charts, 367
Comma-separated value (CSV) file, 418
axes and grid
CSS style, 420
d3.svg.axis() function, 419
grid variables, 421
horizontal grid lines, 422
SVG elements, 421–422
SVG elements addition, 420
data drawing
append() method, 425
axes labels, 424
CSS style settings, 425
d3.csv function, 423
d3.svg.line(), 424
data object, 423
domain, 423
main components, 425
path element, 425
data object, 350
data reading parsing
size and margins, 418
text editor, 419
time format control, 419
filled areas
area chart, 430
line object, 429
path element, 429–430
$.get() function, 345
marks
d3.csv() function, 426
.dot class, 426
jqPlot library, 427
shapes and colors, 427–429
Configuration object, 332
console.log() method, 17
Content delivery network (CDN), 20, 221–222, 275, 283, 289, 312, 329, 449, 576
createDetail() function, 353
createMaster() function, 353–354
CSS Gradient Generator, 47
ctx.lineTo(x,y) function, 74
ctx.strokeStyle() function, 74

D

Data-driven documents (D3) library
append() method, 381, 386
axes range control
limit references, 414–415
variables, 414
axis arrow
CCS style, 416
with and without filling, 416
x and y axes, 415

Data-driven documents (D3) library (*cont.*)
bar charts (*see* Bar charts)
bind arbitrary data, 374
blank HTML page, 374
bubble chart (*see* Bubble chart)
Candlestick charts (*see* Open-high-low-close (OHLC) chart)
CSV file (*see* Comma-separated value (CSV) file)
data() operator, 387
difference line chart
clip paths, 445
code changes, 446
code writing, 441, 443
colors, 444
d3.min and d3.max, 443
data parsing, 443
data point interpolation, 444
generic function, 444
trend area, 445
TSV file, 440
domains and ranges, 402–403
dynamic properties, 390
enter() operator, 388
FIREBUG, 373
first bricks, 401
html() method, 380, 387
input data addition
attr() function, 406
axes, 405
CSS styles, 404–405
grid addition, 407
light gray grid, 408
max(date) function, 403
path element, 404
SVG elements, 403
text(String) function, 405
text style, 408
ticks addition, 406–407
transformation, 403
x and y values, 411–413
insert() method, 384, 386
JavaScript library, 374
multiseries line charts (*see* Multiseries line charts)
operators, 377
PHP language
json() function, 571
multiseries chart, 571
myPHP.php, 569–570
sales.sql, 568
TSV file, 563, 567
pie charts (*see* Pie charts)
radar chart (*see* Radar charts)
real-time charts (*see* Real-time charts)
scales, 402–403
Scatterplot (*see* Scatterplot)
selection, 375
SVG elements (*see* Scalable vector graphics (SVG) elements)
title and axis labels, 417–418
d3.selectAll("selector"), 376
d3.select("selector"), 376
Data-driven documents (D3) library
DateAxisRenderer plug-in, 181–182
DEBUGGING D3 CODE, 373
delay() function, 398
Document object model (DOM) element, 52, 62
Donuts charts
arc creation, 494–495
code writing, 493–494
data array, 261
data_06.csv, 494
data transition, 260
definition, 257
external labels, 496
forEach() function, 495
multiseries, 261–263
path element, 495–496
pieRenderer object, 260
show property, 262
vs. single chart, 492
startAngle property, 260
duration() function, 398

E

each() method, 54
enhancedLegendRenderer plug-in, 333
eq() method, 26

F

forEach() function, 14
Funnel chart
basic, 284
CDN service, 283
data format, 283
fill property, 286
FunnelRenderer, 283
legend and percentages, 285
lineWidth property, 286
sectionMargin property, 285
seriesDefaults object, 283
showDataLabel, 284
without spaces, 285

G

Gantt chart, 367
  bar chart, 364
  categories property, 364–365
  columnrange object, 366
  Date.UTC() function, 364
  highcharts-more.js, 364
  minPadding and maxPadding properties, 365
getJSON() method, 324
Grouped bar charts. *See* Bar charts

H

hide() function, 79
highcharts() function, 332, 347
.Highcharts library
  bar charts, 359
  combined charts (*see* Comma-separated value (CSV) file)
  commercial and noncommerical licenses, 329
  CSV file (*see* CSV file)
  definition, 329
  export, 351
  Gantt chart, 364
  Highstock, 371
  line charts
    alternateGridColor property, 342
    colors band component, 343
    complete line chart, 334
    grid, 338
    input data, 336
    legend, HTML, 341
    marker points customize, 343
    plotBands component, 341
    themes, 344
    tooltip component, HTML, 340
  local method/content delivery network service, 329
  master detail chart, 352
  modules, 330
  MooTools/Prototype, 330
  pie chart, 362
  professional product, 329
  similarities and differences, 331
  themes, 331
highcharts-more.js, 330
Highlighting, line charts
  cursor highlighter
    data point highlighting, 187–188
    options object, 186
    x and y values, 186
  HTML format
    data point highlighting, 188–189
    formatString property, 188
High-low line charts, 208
Highstock, 371
Horizontal bar charts. *See* Bar charts
HTML table, 380, 387
  blank page, 374
  creation, 43
    balloon lost, 43
    color gradation, 51
    CSS gradient generator, 48
    CSS style, 46
  dataGroups, 58
  implementation, 59
  jQuery library, 52
  xLabels
    array, 55
    definition, 54
    extraction (*see* Label extraction)
    log() function, 54

I

Input data handling
  JSON format
    arrays and objects, 320
    $.getJSON() method, 323
    jqPlot Data Renderer, 321
    $.jqplot() function, 324
    multiseries line chart, 324
    value labels, 320
  real-time charts
    doUpdate() function, 326–327
    streaming data, 325, 327
insert() method, 384, 386
Integrated development environment (IDE), 10, 574

J, K

JavaScript
  Aptana Studio, 10–11
  arrays, 13
  IDEs, 10
  libraries, 11
  objects, 15
  online IDE jsFiddle, 11
  running and debugging, 12
  variable, 13
JavaScript test driver (JSTD), 10
jqlotDataClick event, 245–247
jqPlot charts
  accordion, 309
  CSS style settings, 305
   elements, 307
  draggable line charts
    container types, 316
    feature, 315

jqPlot charts (*cont.*)
initial position and size
container, 315
layout container, 317
replot() function, 314
jQueryUI containers, 312
jQueryUI tabs() function, 306
resizable line chart
bind() function, 313
CDN service, 312
chart1, 313
result, 313
results, 307–308
tabs, 303
variables, 305
jqplotDataClick event, 241–242
jqPlotDataHighlight event, 244
jqplotDataMouseOver
event, 243, 245
jqplotDataUnhighlight event, 244
jqPlot framework. *See* Highcharts library
jqplot() functions, 254–255, 305
jqPlot library
basic files, 131–132
CSS customization, 146
data series
array of data, 140
multiple series, 140
multiseries chart, 142
options object, 141
showMarker property, 141–142
single variable, 143
definition, 131
handling options on axes, 138–139
inserting options
chart component, 136
chart structure, 136
chart title, 136
default line chart customization, 135
drawGridlines property, 138
grid lines hiding, 138
grid object, 137
jqPlot object, 136
options object, 137
properties, 135
modules
analysis, 147
ease of maintenance, 147
line chart, 147
myCss.css, 148
myJS.js, 148
new library creation, 131
plot basics
container addition, 133
creation, 133–134
plug-ins, 134–135
renderer
bar chart, 143–144
BarRenderer plugin, 145
CDN service, 143–144
definition, 143
jqPlot distribution, 146
properties, 145
jqPlot Plug-ins, 581
jqplotRightClick event, 242–243
jQuery
DOM elements
new elements, 26
ready() method, 24
remove, hide, and replace
elements, 27
selections, 24
UI libraries
CDN method, 20
chaining methods, 23
graphic elements, 41
hide() method, 23
local method, 20
selection, 21
web server, 21
UI widgets
accordion widget, 29
animation attribute, 37
button, 32
combo box, 34
CSS style settings, 38–39
 element, 38
functionality, 29
Google Hosted library, 28
height and width attributes, 37
menu, 35
progress bar, 39
range attribute, 37
resizing and encapsulation, 29
tab widget, 31
value attributes, 38
jQuery css() function, 83
jQueryUI containers, 312
jQueryUI tabs() function, 306

L

Label extraction
legend array, 56
Math.max.apply() function, 56
tableData\: legend property, 55
tbody td selector, 56
yLabels array, 57
labelFormatter property, 341
Legend, 75

Library creation
default values, 127, 129
features
barGroupMargin property, 117
data, 115
$(document).ready() function, 115–116
HTML table, 115
options, 115
options object, 117–118
target, 115
type property, 117
implementation
axis tick labels, 120, 124
canvas setting, 119–120
data drawing, 124, 126
legend component, 126
jQuery library, 114
myLibrary() function, 113–114
new library, 113
Limit lines
button addition, 191
button types, 191
horizontal limit line, 195
JavaScript functions, 195
jQuery UI widgets, 192–193
lineup() and linedown() function, 193
lower and upper thresholds, 194
vertical lines, 194
CanvasOverlay plug-in
horizontalLine and dashedHorizontalLine, 190
lineCap property, 190
lower and upper limits, 190
Line charts
area charts (*see* Area charts)
axes (*see* Axes, line charts)
band charts, 208, 213
canvas (*see* Canvas)
cartesian chart, 66
chart appearance, CSS
background color, 201
chart-container class, 202
 element, 202
file extraction, 200
grid setting, 203, 205
multiseries line chart, 203
options object, 200
tick labels and title, 201
closePath() function, 74
ctx.lineTo(x,y) function, 74
ctx.strokeStyle() function, 74
dataGroups, 74
data point, 66
date values
DateAxisRenderer plug–in, 180, 182
day-by-day point values, 183
different x input value formats, 183
formatString property, 183
fillBetween object, 213, 216
grid lines, 71
hide tables, 79
highlighting (*see* Highlighting, line charts)
JavaScript code and CSS styles, 75
JavaScript data, 176
math functions, 176, 178
random data generation, 179
legend, 75
limit lines (*see* Limit lines)
lineWidth property, 73
Log scale
exponent, 165
renderer property, 165
semilog scale, 166
values, 165
multiseries (*see* Multiseries line chart)
tick labels
CSS style class, 70
Firebug menu, 67
 tags, 66
unordered lists, 70
x axis, 66
xLabels and yLabels arrays, 68
time values, 184
title, 77
trend lines, 217, 220
x, y values, 151
indexes of passed array, 151
linear plot, 152
nonuniformly distributed points, 152
zooming
cursor plug-in, 196
incoming data availability, 197
line chart extraction, 198
resetZoom() method, 198
zoom property, 197
lineWidth property, 73
log() function, 373
Log scale
exponent, 165
renderer property, 165
semilog scale, 166
values, 165

M, N

Marimekko chart, 238–240
Master detail chart
bottom and foreground chat, 358
createDetail() function, 353–354
createMaster() function, 353–354

Master detail chart (*cont.*)
detailOptions, 353
large amount data, 352
library files and the dark green theme, 353
masterOptions object, 354
min and max values, 356
options, 357
two charts, 352
Math.max.apply() function, 56
maxPadding property, 365
Mekko chart, 238–240
minPadding property, 365
Multiseries line chart, 166
animated charts, 173
D3 chart
code writing, 431, 434
color setting, 434
d3.keys() function, 435
data_02.tsv, 431
data map function, 435
double iteration, 436
legend addition, 437–438
line interpolation, 439–440
parseDate() function, 434
SVG element, 436
data array
customized color set, 168–169
multiple arrays, 166
transparency levels, 169
line and marker style
line pattern and width, 171–172
linePattern property, 172
more than one y axis, 174–175
smooth–line chart, 169–170

O

Object configuration, 333
Open-high-low-close (OHLC) chart, 6
box representation, 511–512
creation
code writing, 504
CSS styles, 508
data_08.csv, 503
data parsing, 505
forEach() function, 506
jqPlot, 508
line element, 507
minimum and maximum values, 506
SVG elements with axes and labels, 506
data format, 509

P, Q

parseInt() function, 38
Pie charts, 362
canvas setting
code status, 85, 88
strokeRect(), 88
caption selection, 92
color sequence, 487–488
CSS attributes, 92–93
data array, 257
dataLabels property, 258
data representation, 1, 94
definition, 257
drawing
arc.centroid() function, 485
arc element, 484
area and margins, 482
circular sector, 482
colors, 482
CSS class attributes, 486
d3.csv() function, 484
d3.layout.pie() function, 483
data_04.csv, 482
firebug, 485
outer and inner radius, 483
root element insertion, 483
simple pie chart, 486
title addition, 485
element addition, 92
gradient effect
black gradient and space divisions, 96
hide() function, 95
sliceGradientColor and borderstyle, 95–96
slices, 94–95
implementation
center point variables, 89
counter and fraction, 91
dataSum function, 89–90
distance variable, 91
HTML structure generation, 91
 tag, 91
pieMargin, 89
sliceMiddle variable, 91
slices, 90
lineWidth property, 260
mixing, 491–492
multilevel, 263–264
percentage report, 258
PieRenderer plug-in, 257
plug-in activation, 257
pulled out slice, 98–99
animation, 99, 104
code writing, 97–98

counter addition, 108
fluctuating animation, 112
for() loop function, 104
handleChartClick() event, 105–106
handleChartClick() function, 111
new slice() function, 104
nextMove and k variables, 105
Out and ins variables, 107
pullout() function, 109
pushIn() function, 110
setInterval() functions, 109
slice extraction, 108
startAngle and endAngle slice, 97
sliceMargin property, 259
slices, 85
slices only with outlines, 490
sorting, 488–489
space addition, 489
SVG elements, 481
plotBands component, 341
plotOptions, 337
pointInterval, 337
Polar area diagrams
arcs, 499
code writing, 497–498
CSS style classes, 500
d3.csv() function, 498
data_07.csv, 497
legend, 500
radius and angle, 497
slices with labels, 499
Portable network graphics (PNG) file, 253

R

Radar charts
auto scaling axes
area and margins, 546
circle-ticks, 548
circular grid, 548
d3.csv() function, 547
data_11.csv file, 546–549
drawing area center, 546
linear scale, 546
radial axes, 549–550
root element <svg>, 546
<svg> elements, 547
ticks, 547
data addition
color domain addition, 550
CSS style class addition, 553
d3.csv() function, 554
data_12.csv, 553
data series, 551
legend addition, 551
path color, 554
semi-transparent areas, 555
web structure, 545
ready() method, 24
Real-time charts
CSS styles, 560
data array, 558
getRandomInt() function, 561–562
HTML structure, 557
line() function, 559–560
margin adjustment, 558
requestAnimFrame() function, 562
SVG elements, 558–559
updateData() function, 561
replaceWith() method, 28
replot() function, 309, 314
resizable() function, 313

S

Scalable vector graphics (SVG) elements, 7, 401
create, 391
transformations
rotation, 396
scale, 395
translate, 395
transitions, 397
Scatter chart
CDN service, 275
data collection, 274
default settings, 273–274
definition, 273
trend line, 275
Trendline plug-in, 275
Scatterplot
axis label, 516
cluster analysis
algorithm implementation, 530
circle SVG elements, 536
code writing, 533
K-mean algorithm, 530-533
kmean() function, 535
myPoints, 535
representation, 536
selection and grouping criteria, 529
code writing, 514
colored markers, 517
customized markers
cell markers, 523
group markers, 523
set of markers, 523
SVG editor, 522
SVG element path, 523
d3.tsv() function, 516
data_09.tsv, 513–514

Scatterplot (*cont.*)
data distribution, 518
data points highlighting, 537–540
markers and symbols
CSS styles, 520
groupMarker object, 520
line replacement, 520
predefined symbols, 519
series representation, 521
nice() function, 516
SVG element, 517
trendlines
code writing, 524
data parsing, 527
least square method, 524
representation, 528
summations, 528
tsv() function, 527
select() method, 376
slicedOffset property, 362
Smooth-line chart, 169
Stacked bar chart
drawing
code writing, 456, 458
color domain, 458
data_05.csv, 456
data() function, 461
Firebug console, 459–460
forEach() function, 458
iterative function, 459
numeric values, 458
rect elements, 461
text element, 462
translate(x,0) function, 461
x and y domain, 460
normalized
code addition, 467
d3.csv() function, 465
data_05.csv file, 464
legend, 466
new style class addition, 467
range of values, 464
rotate() function, 465
SVG elements, 468
title modification, 465
strokeRect() function, 64
style() operator, 379

T

Tab-separated value
(TSV) file, 564, 568
text() operator, 377
tickRenderer property, 223
transition() method, 398
translate() method, 63
Trend lines, 217, 220

U, V

useHTML property, 341

W

Web Graphics Library (WebGL), 7
World Wide Web Consortium (W3C), 9

X, Y, Z

XAMPP and Aptana Studios, 573
IDE installation, 574
project creation, 575
run configuration, 579
src directory
D3 library version 3, 577
files and subdirectories, 578
highcharts library version 3.0.5, 577
jqPlot library version 1.0.8, 577
jquery UI library version 1.10.3, 577
web server installation, 573
workspace
charts directory, 576
setting, 574–575
src directory, 576
xLabels
array, 55
definition, 54
label extraction
legend array, 56
Math.max.apply() function, 56
tableData:legend property, 55
tbody td selector, 56
yLabels array, 57
log() function, 54
XY chart. *See* Scatter chart

Beginning JavaScript Charts

With jqPlot, D3, and Highcharts

Fabio Nelli

Apress®

Beginning JavaScript Charts

Copyright © 2013 by Fabio Nelli

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.

ISBN-13 (pbk): 978-1-4302-6289-3

ISBN-13 (electronic): 978-1-4302-6290-9

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

President and Publisher: Paul Manning
Lead Editor: Ben Renow-Clarke
Development Editors: James Markham and Chris Nelson
Technical Reviewer: Matthew Canning
Editorial Board: Steve Anglin, Mark Beckner, Ewan Buckingham, Gary Cornell, Louise Corrigan, Jim DeWolf, Jonathan Gennick, Jonathan Hassell, Robert Hutchinson, Michelle Lowman, James Markham, Matthew Moodie, Jeff Olson, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Gwenan Spearing, Matt Wade, Steve Weiss
Coordinating Editor: Jill Balzano
Copy Editors: Lisa Vecchione, Kezia Endsley, and Brendan Frost
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global
Cover Designer: Anna Ishchenko

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this text is available to readers at www.apress.com/9781430262893. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/.

This book is dedicated to my grandfather Polo and my grandmother Franca,
for all the support they have given me in life

Contents

About the Author xix

About the Technical Reviewer xxi

Acknowledgments xxiii

Introduction xxv

Chapter 1: Charting Technology Overview 1

Elements in a Chart 1

Most Common Charts 4

How to Realize Charts on the Web 6

HTML5 6

Charting with SVG and CANVAS 7

Canvas vs SVG 7

The DOM 9

Developing in JavaScript 10

Running and Debugging JavaScript 12

Data Types in JavaScript 13

Firebug and DevTools 16

JSON 18

Summary 18

Chapter 2: jQuery Basics 19

Including the jQuery Library 20

Selections 21

Chaining Methods 23

The Wrapper Set 23

jQuery and the DOM 24
The ready() Method 24
Traversing the DOM with Selections 24
Create and Insert New Elements 26
Remove, Hide, and Replace Elements 27
jQuery UI: Widgets 28
Accordion 29
Tab 31
Button 32
Combo Box 34
Menu 35
Slider 37
Progress Bar 39
Concluding Thoughts on the jQuery Library 41
Summary 41
Chapter 3: Simple HTML Tables 43
Creating a Table for Your Data 43
Your Example's Goals 43
Applying CSS to Your Table 46
Adding Color Gradation to Your Table 47
Adding Color Gradation to Your Table, Using Files 51
Parsing the Table Data 52
Importing the jQuery Library 52
xLabels 53
dataGroups 58
Ready for Implementing Graphics 59
Summary 59

Chapter 4: Drawing a Line Chart 61
Defining the Canvas 61
Setting the Canvas 63
Drawing a Line Chart 65
Drawing Axes, Tick Labels, and the Grid 66
Drawing Lines on the Chart 73
Adding a Legend 75
Adding a Title 77
Hiding the Table 79
Summary 80
Chapter 5: Drawing a Bar Chart 81
Drawing a Bar Chart 81
Summary 84
Chapter 6: Drawing a Pie Chart 85
Drawing a Pie Chart 85
Setting the Canvas 85
Implementing the Pie Chart 89
Completing the Pie Chart 92
Adding Effects 94
Adding a Gradient Effect 94
Adding a Better Gradient Effect 96
Creating a Pie Chart with a Slice Pulled Out 97
Inserting an Animation to Pull Out the Slice 99
Clicking a Slice to Pull It Out 104
Clicking a Slice to Pull It Out with Animation 108
Other Effects 112
Summary 112

Chapter 7: Creating a Library for Simple Charts 113

Creating a Library 113

Main Features: Target, Data, and Options 114

Implementing the Library 119

Setting the Canvas 119

Drawing the Axes, Tick Labels, and Grid 120

Drawing Data 124

Adding the Legend 126

Default Values 127

Summary 130

Chapter 8: Introducing jqPlot 131

The jqPlot library 131

Including Basic Files 131

Plot Basics 133

Adding a Plot Container 133

Creating the Plot 133

Using jqPlot Plug-ins 134

Understanding jqPlot Options 135

Inserting Options 135

Handling Options on Axes 138

Inserting Series of Data 140

Renderers and Plug-ins: A Further Clarification 143

CSS Customization 146

Thinking in Modules 147

Summary 149

Chapter 9: Line Charts with jqPlot 151

Using (x, y) Pairs as Input Data 151

First Steps in the Development of a Line Chart: The Axes 153

Add a Title and Axis Labels 153

Axis Properties 155

Axes Ticks 158

Using the Log Scale 164

The Multiseries Line Chart 166

Multiple Series of Data 166

Smooth-Line Chart 169

Line and Marker Style 171

Animated Charts 173

More Than One y Axis 174

Data with JavaScript 176

Generating Data, Using Math Functions 176

Generating Random Data 179

Handling Date Values 180

The *DateAxisRenderer* Plug-in 180

Handling Date Values in Different Formats 183

Handling Time Values 184

Highlighting 185

Cursor Highlighter 186

Highlighting with HTML Format 188

Interacting with the Chart: Limit Lines and Zooming 189

Drawing a Limit Line on the Chart 189

Adding Buttons to Your Charts 191

Zooming 196

Changing Chart Appearance 199
Customizing Text, Using CSS 200
Changing the Background Color 201
Further Customization, Using CSS 202
Setting the Grid 203
Working with Areas on Line Charts 205
Area Charts 206
Line and Area Charts 208
Band Charts 208
Filling Between Lines in a Line Chart 213
Trend Lines 217
Summary 220
Chapter 10: Bar Charts with jqPlot 221
Using the *BarRenderer* Plug-In to Create Bar Charts 221
Rotate Axis Tick Labels 223
Modify the Space Between the Bars 224
Adding Values at the Top of Bars 225
Bars with Negative Values 226
Bar Charts with More Than One Set of Data 228
Vertical and Horizontal Bar Charts 230
Vertical Stacked Bars 232
Horizontal Stacked Bars 234
Combination Charts: Lines in Bar Charts 235
Animated Plot 237
Marimekko Chart 238
Bar Chart Events 240
The jqplotDataClick Event 241
The jqplotRightClick Event 242
Other Bar Chart Events 243
Clicking the Bar to Show Information in Text 245

Handling Legends....247
Adding a Legend....247
The Enhanced Legend....250
Custom Legend Highlighting....251
Custom Tool Tip....253
Summary....256
Chapter 11: Pie Charts and Donut Charts with jqPlot....257
Pie Charts....257
Donut Charts....260
Multilevel Pie Charts....263
Summary....265
Chapter 12: Candlestick Charts with jqPlot....267
OHLC Charts....267
Using Real Bodies and Shadows....270
Comparing Candlesticks....270
Summary....272
Chapter 13: Scatter Charts and Bubble Charts with jqPlot....273
Scatter Chart (xy Chart)....273
Bubble Chart....276
Block Chart....279
Summary....281
Chapter 14: Funnel Charts with jqPlot....283
Creating a Funnel Chart....283
Summary....286
Chapter 15: Adding Controls to Charts....287
Adding Controls....287
Using Radio Buttons....288
Adding Radio Button Controls....288
Accessing Attributes after the Chart Has Already Been Drawn....292

Using Sliders 294
Using Check Boxes 298
Summary 301

Chapter 16: Embedding jqPlot Charts in jQuery Widgets 303
jqPlot Charts on Tabs 303
jqPlot Charts on Accordions 309
Resizable and Draggable Charts 312
A Resizable Line Chart 312
Three Draggable Line Charts 314
Summary 317

Chapter 17: Handling Input Data 319
Using the JSON Format 319
The JSON Format 320
A Practical Case: The jqPlot Data Renderer 321
JSON and $.getJSON() 323
Real-Time Charts 325
Summary 328

Chapter 18: Moving from jqPlot to Highcharts 329
The Highcharts Distribution 329
Similarities and Differences 331
Line Charts with Highcharts 333
Completing the Line Chart 334
Different Ways of Handling Input Data 336
The grid: Advanced Management 338
Customizing Tooltips with HTML 340
Customizing the Legend with HTML 341
Adding Bands 341
Customizing the Marker Points 343
The Themes of Highcharts 344

Reading Data from a File 345
Reading a CSV File Using $.get() 345
Excluding CSV Columns from Your Data 350
Exporting the Chart 351
The Master Detail Chart 352
Bar and Pie Charts with Highcharts 359
Bar Charts 359
Pie Charts 362
Gantt Charts 364
Combined Charts 367
Highstock Library 371
Summary 372
Chapter 19: Working with D3 373
Introducing D3 374
Starting with a Blank HTML Page 374
Using Selections and Operators 375
Selections and Selectors 375
Operators 377
Creating New Elements 380
The html() Method 380
The append() Method 381
The insert() Method 384
Inserting Data into Elements 387
Applying Dynamic Properties 390
Adding SVG Elements 391
Creating an SVG Element 391
Transformations 393
Transitions 397
Summary 399

Chapter 20: Line Charts with D3 401

Developing a Line Chart with D3 401

Starting with the First Bricks 401

Scales, Domains, and Ranges 402

Inside the Code 403

Using Data with (x, y) Values 411

Controlling the Axes' Range 414

Adding the Axis Arrows 415

Adding a Title and Axis Labels 417

Drawing a Line Chart from Data in a CSV File 418

Reading and Parsing Data 418

Implementing Axes and the Grid 419

Drawing Data with the csv() Function 423

Adding Marks to the Line 426

Line Charts with Filled Areas 429

Multiseries Line Charts 431

Working with Multiple Series of Data 431

Adding a Legend 437

Interpolating Lines 438

Difference Line Chart 440

Summary 448

Chapter 21: Bar Charts with D3 449

Drawing a Bar Chart 449

Drawing a Stacked Bar Chart 455

A Normalized Stacked Bar Chart 464

Drawing a Grouped Bar Chart 468

Horizontal Bar Chart with Negative Values 475

Summary 479

Chapter 22: Pie Charts with D3 481
The Basic Pie Charts 481
Drawing a Basic Pie Chart 482
Some Variations on Pie Charts 487
Donut Charts 492
Polar Area Diagrams 497
Summary 501
Chapter 23: Candlestick Charts with D3 503
Creating an OHLC Chart 503
Date Format 509
Box Representation in Candlestick Charts 511
Summary 512
Chapter 24: Scatterplot and Bubble Charts with D3 513
Scatterplot 513
Markers and Symbols 519
Using Symbols as Markers 519
Using Customized Markers 521
Adding More Functionalities 524
Trendlines 524
Clusters 529
Highlighting Data Points 537
Bubble Chart 541
Summary 543
Chapter 25: Radar Charts with D3 545
Radar Chart 545
Building Auto Scaling Axes 546
Adding Data to the Radar Chart 550
Improving Your Radar Chart 553
Summary 556

Chapter 26: Handling Live Data with D3 557
Real-Time Charts 557
Using PHP to Extract Data from a MySQL Table 563
Starting with a TSV File 564
Moving On to the Real Case 568
Summary 571
Conclusion 571
Appendix A: Guidelines for the Examples in the Book 573
Installing a Web Server 573
Installing Aptana Studio IDE 574
Setting the Aptana Studio Workspace 574
Creating a Project 575
Completing the Workspace 576
Filling the src Directory with the Libraries 577
Running the Examples 578
Summary 579
Appendix B: jqPlot Plug-ins 581
Index 583

About the Author

Fabio Nelli is an information technology scientific application specialist at IRBM Science Park, a private research center in Pomezia, Italy. He was a computer consultant for many years at IBM, EDS, and Merck Sharp and Dohme, along with several banks and insurance companies. He worked as well as a specialist in information technology and automation systems at Beckman Coulter.

He holds a Master's degree in Organic Chemistry from La Sapienza University of Rome. He recently earned a Bachelor's degree in Automation and Computer Engineering from eCampus University of Novedrate.

Nelli is currently developing Java applications that interface Oracle databases, using scientific instrumentation to generate data, and web server applications that provide analysis to researchers in real time.

Web site: www.meccanismocomplesso.org

About the Technical Reviewer

Matthew Canning is an author, speaker, and experienced technical leader who has served in engineering and management roles at some of the world's largest companies. Aside from technology, he writes and presents on subjects such as memory, mental calculation, and productivity. He currently lives outside Philadelphia with his wife and daughter.

Twitter: @MatthewCanning
Web site: matthewcanning.com

Acknowledgments

I would like to express my gratitude to all the people who played a part in developing this book. First, a special thanks to Ben Renow-Clarke for giving me the opportunity to write the book. Thanks to Jill Balzano and Mark Powers for their guidance and direction. Thanks also to everyone who took part in the review and editing of the book for their professionalism and enthusiasm: Chris Nelson, Matthew Canning, James Markham, Lisa Vecchione, Kezia Endsley, Brendan Frost, and Dhaneesh Kumar.

Finally, a thank-you to Victor Jacono for his invaluable help with the English text.